HZ BOOKS
华章教育

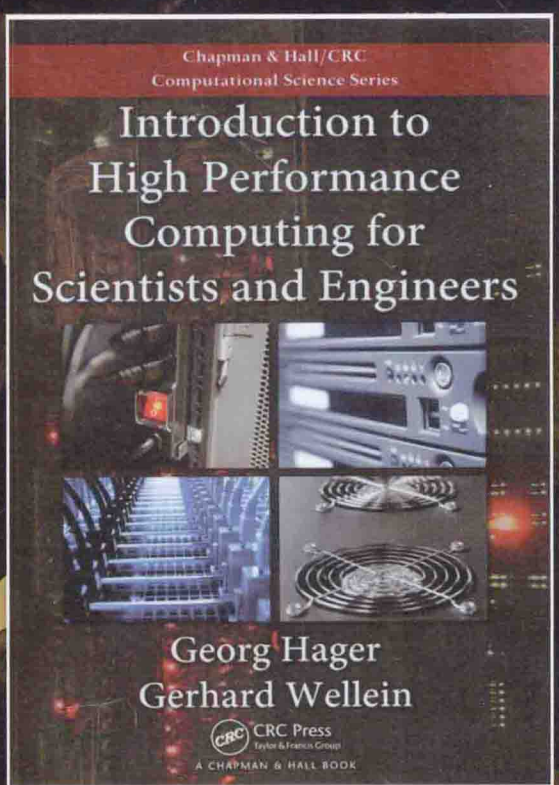
计 算 机 科 学 丛 书

CRC Press
Taylor & Francis Group

高性能科学与工程计算

（德）Georg Hager Gerhard Wellein 著
张云泉 袁良 贾海鹏 张先轶 译

Introduction to High Performance Computing for Scientists and Engineers



机械工业出版社
China Machine Press

高性能科学与工程计算

Introduction to High Performance Computing for Scientists and Engineers

Georg Hager和Gerhard Wellein在本书中介绍了高性能计算的基础知识，写作和描述方式十分易读。……本书全面讲解了理论、技术、体系结构、现代高性能计算机软件实现以及高性能计算系统使用等知识，重点关注科学与工程计算中的问题，具有教育性和独特性，我向科学家和工程师强烈推荐此书。我相信本书能使许多读者收益良多并为他们提供一个优秀参考。

——Jack Dongarra，田纳西大学教授，美国国家工程院院士

本书由高性能计算专家编写，介绍了当前主流的计算机体系结构、主流并行编程模型、常用优化策略。本书对程序性能方面的讲解采用直观的方式，不需要丰富的计算机科学知识就能理解。本书可作为读者学习更高级知识的基础。

本书特点

- 覆盖基本的串行优化策略和主流并行模式。
- 强调在不同系统体系结构上程序性能建模的重要性。
- 包含大量作者在多年用户支持、程序优化和评测工作中积累的实例。
- 介绍了一些重要概念，例如多核体系结构和亲缘性。
- 展示了大量Fortran、C和C++代码实例。

作者简介

Georg Hager 是一位理论物理学家，拥有Greifswald大学计算物理博士学位。他从1995年致力于高性能计算研究，现在是Erlangen Regional Computing Center (RRZE)高性能计算组的高级科学家。最近研究兴趣包括现代微处理器的体系结构优化、处理器和系统级的性能建模以及异构系统优化。他的日常工作涉及高性能计算的全方位用户支持，例如讲座、报告、培训、代码并行化、代码分析和优化以及新型计算机体系结构和工具的评估。

Gerhard Wellein 拥有Bayreuth大学固体物理博士学位，现在是Erlangen大学计算机系教授。他是Erlangen Regional Computing Center (RRZE)高性能计算组的主任，有十余年的高性能计算教学经验，其教学对象包括计算科学和工程编程领域的学生和科学家。他的研究兴趣包括大规模稀疏特征值问题求解、新奇并行化方法、性能建模以及体系结构优化方法。

CRC Press
Taylor & Francis Group

投稿热线：(010) 88379604
客服热线：(010) 88378991 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn

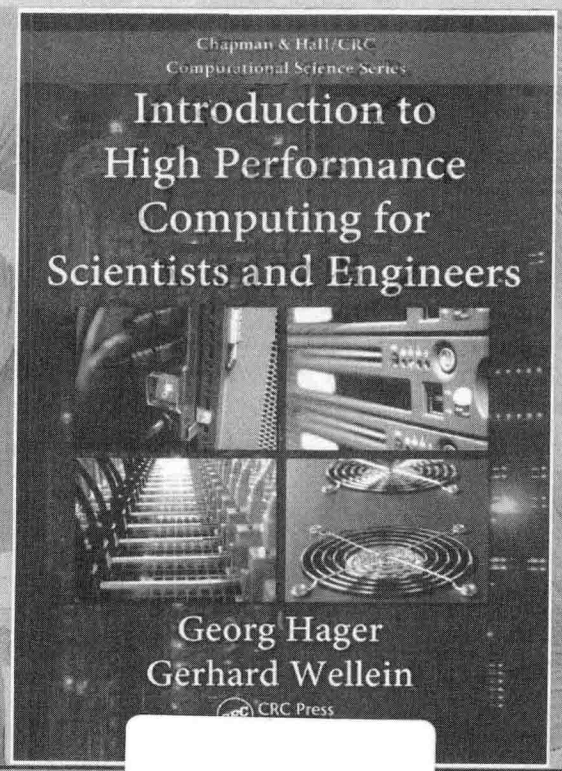
封面设计：锋 杉

计 算 机 科 学 丛

高性能科学与工程计算

（德）Georg Hager Gerhard Wellein 著
张云泉 袁良 贾海鹏 张先轶 译

Introduction to High Performance Computing for Scientists and Engineers



机械工业出版社
China Machine Press

图书在版编目（CIP）数据

高性能科学与工程计算 /（德）海格（Hager，G.），（德）韦雷因（Wellein，G.）著；张云泉等译 . —北京：机械工业出版社，2014.6
（计算机科学丛书）
书名原文：Introduction to High Performance Computing for Scientists and Engineers

ISBN 978-7-111-46652-9

I. 高… II. ① 海… ② 韦… ③ 张… III. 工程计算 IV. TB115

中国版本图书馆 CIP 数据核字（2014）第 093761 号

本书版权登记号：图字：01-2013-5984

本书主要面向科学家、研究者和工程师，采取一种循序渐进的方式介绍高性能计算的基础知识，即使不熟悉计算科学的读者也非常容易理解。本书主要覆盖了当代处理器体系结构基础知识、各种基本的程序串行优化策略和基本并行模式。此外，在并行编程模型方面，还包括基于 OpenMP 的共享并行和基于 MPI 的分布式并行编程，并包含大量 Fortran 代码实例，方便读者学习理解。总之，本书以一种独特的视角介绍高性能计算知识，适合各应用领域的专家和学生研读。

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码：100037）
责任编辑：朱　劼　　肖晓慧　　责任校对：董纪丽
印　　刷：北京瑞德印刷有限公司　　版　　次：2014 年 7 月第 1 版第 1 次印刷
开　　本：185mm × 260mm　1/16　　印　　张：16
书　　号：ISBN 978-7-111-46652-9　　定　　价：69.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：(010) 88378991　88361066　　投稿热线：(010) 88379604
购书热线：(010) 68326294　88379649　68995259　　读者信箱：hzjsj@hzbook.com

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章公司较早意识到“出版要为教育服务”。自 1998 年开始，我们就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与 Pearson，McGraw-Hill，Elsevier，MIT，John Wiley & Sons，Cengage 等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出 Andrew S. Tanenbaum，Bjarne Stroustrup，Brain W. Kernighan，Dennis Ritchie，Jim Gray，Afred V. Aho，John E. Hopcroft，Jeffrey D. Ullman，Abraham Silberschatz，William Stallings，Donald E. Knuth，John L. Hennessy，Larry L. Peterson 等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专程为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近两百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

华章网站：www.hzbook.com
电子邮件：hzjsj@hzbook.com
联系电话：（010）88379604
联系地址：北京市西城区百万庄南街 1 号
邮政编码：100037

华章教育
华章科技图书出版中心

译 者 序

Introduction to High Performance Computing for Scientists and Engineers

以高性能计算为核心的计算科学，已经广泛应用于科学研究和工程设计的各个层面，成为继理论研究和实验研究之外的第三种科学研究方法。美国总统信息技术顾问委员会在2005年发表题为“计算科学：确保美国竞争力”的报告中认为计算科学是21世纪最重要的技术领域之一，它对于整个社会的进步是不可或缺的，并建议美国政府制定规划，对计算科学进行长期资助。

高性能计算追求程序执行效率，因此并行程序的实现与优化成为计算机科学家的两大研究主题。本书也以并行编程模型和并行优化为主要内容，对程序串行优化、数据优化和并行性优化做了深入介绍，还对多处理器导致的NUMA问题以及多核环境下的进程亲缘性进行了深入讲解。在并行编程模型方面，OpenMP和MPI分别是共享存储系统和分布式存储系统两大并行计算机体系架构上的事实标准。本书不仅详细介绍了这两种编程语言，并且还单独设立章节讲解了相应的关键优化技术。此外，本书配备大量程序和命令实例，方便读者掌握相关技巧。总之，本书作者通过自己多年的高性能计算领域编程和优化的实践经验，选取最为关键的基本知识和优化技术进行讲解，可供相关领域科学家作为参考。

本书作者Georg Hager和Gerhard Wellein均为物理学家，长期在Erlangen Regional Computing Center (RRZE)高性能计算组从事高性能计算研究，Gerhard Wellein目前还担任该小组主任，主要研究兴趣包括现代微处理器的体系结构相关优化方法、处理器和系统层次的性能模型，以及异构系统优化。两位作者在高性能计算领域都有丰富的全方位用户支持和教学经验，例如讲座、报告、培训、代码并行化、代码分析和优化以及新颖计算机体系结构和工具的评估，其教学对象包括计算科学和工程编程领域的学生和科学家。

由于时间仓促，加上书中某些术语目前没有统一译法，所以我们对一些术语采取了保留其英文名称的方法。对书中翻译方面的错误和不妥之处，恳请广大读者不吝批评指正。

Georg Hager 和 Gerhard Wellein 在本书中为科学家和工程师循序渐进地介绍了高性能计算知识。本书的风格和讲解方法十分易读并容易理解。

二十多年前提出的计算建模和模拟的概念已经成为理论研究和实验研究之外的第三种科学研究方法，软件、数学模型和算法是科学计算领域的关键核心。并行硬件的发展十分迅猛，特别是以指数速度增长的处理器性能以及处理器体系结构和超级计算机系统设计的研究。当计算建模和模拟成为科学研究的第三种方法后，复杂软件及其生态系统将会处于本研究领域的核心地位。

在应用层，科学必须展现为数学模型，并进一步表达为算法和对应的软件代码。相应地，大部分科学基金也在转向这样的项目，即需要领域科学家、计算机科学家及应用数学家协同合作，从原始科学构想到最终可执行软件的研究。这样的项目也需要数学库、协议和系统软件等这些需要花费数年开发并需要持久维护的大规模底层架构的支持，这些软件通常比最初设计的硬件平台甚至是设计和开发人员存活更长时间。

本书覆盖了当代处理器体系结构的基础知识，以及为科学计算程序有效地利用硬件特性进行串行优化的基本技术。作者讨论了数据移动中的关键问题并辅以实例，并以简单易读的方式介绍了高性能计算中的重要问题。书中还讨论了共享存储、非一致访问以及分布式存储的并行化方法。除此之外，还重点介绍了常用的并行编程模型，例如 OpenMP、MPI 以及混合编程方法。

我们生活在一个高效利用超级计算机系统进行大规模高性能计算的时代。本书对并行理论、优化技术、体系结构和现代高性能计算系统软件等方面进行了介绍，特别是对科学和工程问题的关注使得本书成为一本独特的教材，因此我强烈向科学家和工程师推荐此书，本书可以作为一本优秀的参考书，我相信本书会令大多数读者受益。

Jack Dongarra
美国田纳西大学

前　言

Introduction to High Performance Computing for Scientists and Engineers

当1941年构建世界上第一台全自动可编程并具备二进制浮点运算能力的计算机时[H129]，Konrad Zuse成功地预见了这种革命性设备不仅只应用于科学和工程领域，还将对生活的各个方面产生深远影响[H130]。计算机的高效运算、可视化和数据处理能力，允许我们自动执行大量任务并实现无延迟通信，这使得Zuse的预言成为现实，计算已经彻底地改变了我们的生活方式和科研手段。

科学和工程研究从另一种特别的角度受益于计算能力的增长。早期的科研人员已经意识到计算机可以通过虚拟实验代替现实中那些太过复杂、昂贵或极端危险的真实实验，或者进行以前无法手工完成的计算，计算流体力学（Computational Fluid Dynamics，CFD），即模拟任意几何形状的流体就是一个这样典型的应用。飞机、汽车、高速列车以及涡轮设备的设计都离不开流体力学分析。以前的风洞和木质实体模型属于实验研究，而在包括流体力学在内的几乎所有科学研究领域，计算已成为理论分析和实验设计之外的第三种科学研究方法。近几年药物设计已成为高性能计算的新兴研究方向，化学家可以利用软件（只需点击鼠标）快速发现化学反应机制，模拟影响生命运行机制的大分子间的复杂动力学。理论固体物理通过在量子级别对组成成分、原子核和电子的相互作用进行建模来在微观级别研究物质结构[A79]，需要的大量自由度使其无法在理论层次上进行分析，而只能借助大规模计算资源实现。其他同样需要大规模计算的学科包括量子化学、物质科学、结构力学和医学图像处理等。

计算机模拟已成为学术和工业界大多数研究领域中不可或缺的标准工具，虽然科研人员可以利用个人计算机处理许多计算任务，但是仍有一些任务需要个人计算机无法满足的大规模存储、内存以及计算速度，而这正是高性能并行计算机系统的用武之地。

利用高性能计算（High Performance Computing，HPC）作为科研工具至少需要了解相关硬件和软件知识，即使利用具有良好界面可直接运行的软件也是如此，当需要编写代码时掌握这些知识就显得更为必要。但是我们与科学家和工程师多年的合作经验表明，很难在不同研究组中建立和维护充足、完整的相关知识。陡峭的高性能计算学习曲线使得新的博士生难以独自快速掌握相关知识。虽然高性能计算具有基础性、难以掌握且极为昂贵等特点，但是高性能计算毕竟只是一个工具，而推进科学研究的进展才是终极目标。技术的进步使得高性能计算从院系级别扩展到了桌面级别，在当前单处理器发展停滞以及通过增加并行性来提高性能的趋势下，大量科学家和工程师必须关注性能和可扩展性，这也是本书的主题。我们编写此书的目的为保持这些知识的新颖性。

事实上，目前已经出版了多种计算机体系结构、优化和高性能计算方面的书籍[S1，R34，S2，S3，S4]，虽然基本原理是一致的，但是由于向量处理器的衰落、无处不在的SIMD处理能力、多核处理器的出现、ccNUMA存储结构的发展以及高效能高性能网络互连系统的出现，这些书中的许多内容都已过时。发展最为迅猛的要数运行Linux操作系统的基于x86结构的Intel或AMD处理器商业集群。最新的出版物更关注某个特定方面，因此不适合作为学生教材或者相关科学家参考。本书的目标是从性能角度介绍体系结构和高性能计算编程的基础知识。我们的经验表明，大多数用户经常无法定位性能因素以及是否需要考虑优化。本书的读者可以掌握如何确定性能限制因素等基础知识，这为他们学习更为专业的技巧奠定了基础。因此本书也提供了一个详尽的参考书目列表，可以在本书的网页中（http://

www.hpc.rrze.uni-erlangen.de/HPC4SE）找到相应的具有超链接和附加评论的版本。

本书读者

在科学计算中心的多年工作经验使我们熟知用户和并行计算机厂商的需求，因此从事与高性能计算相关工作的人员都会从本书中获益。计算机科学、计算工程或更广泛的关心模拟领域的教师和学生可将本书作为教材。对于希望快速了解高性能计算基础知识的科学家和工程师，本书可以作为入门级参考书。最后，集群系统构建师可以利用本书的内容优化机器，为用户提供更好的服务。读者需要具有一定的编程和高级计算机体系结构知识。需要强调的是，本书只是一本入门级教材而不是一本详尽的参考书，因为到目前还没有一本高性能计算百科全书。

本书内容

高性能计算涉及给定算法（或者称为程序代码）的实现以及所运行的硬件平台。我们假定希望利用高性能计算资源的用户已经充分了解他们想要解决的问题的不同算法，因此我们并不全面介绍它们。本书中我们选取某些特定实例进行讲解，但是可能存在其他更为合适的算法，读者需要理解本书实例中的策略。

虽然我们努力保持本书的精简性，但是不可避免地包含了过多的内容。由于技术发展极为迅猛，我们忽略了一些近期的研究成果，例如现代加速技术（GPGPU、FPGA、Cell 处理器），因为这些内容很快将会变得过时。有些人认为高性能输入、输出技术应该包含在高性能计算书籍中，但是我们认为高效并行 I/O 技术是一个高级并且与硬件系统相关的课题，所以最好单独介绍。在软件方面，我们介绍了基本的程序串行优化策略和基本并行模式：基于 OpenMP 的共享存储并行化、基于 MPI 的分布式存储并行编程。其他一些模式，包括 Unified Parallel C（UPC 语言）、Co-Array Fortran（CAF 语言）和其他一些现代编程方法，仍然需要检验其高效性，就像被广泛接受的 MPI 和 OpenMP 语言一样。

虽然我们不能忽视商业系统的统治性地位，但本书中许多概念都在与特定的体系结构无关的层次上进行介绍，因此当我们应用实例介绍并展示实际性能数值时，都是基于 x86 集群系统标准网络连接。几乎所有的代码都用 Fortran 编写，仅当需要与外界环境相关的特性时我们才用 C 或者 C++ 代码。本书中一些用来测试的代码可以在本书官方网站上下载：http://www.hpc.rrze.unierlangen.de/HPC4SE。

本书按下述方式组织：第 1 章介绍现代基于高速缓存的微处理器体系结构，并讨论其内在性能瓶颈，对最新的发展（例如多核芯片和同时多线程（SMT））也进行了介绍。虽然向量处理器已不再在高性能计算市场中应用，但是我们还是对其进行了简要介绍。第 2 章和第 3 章讲解了基于高速缓存体系结构的串行程序的一般优化方法，介绍了用来寻找循环程序最优性能的一些简单的模型，还展示了如何通过代码变换来改进其性能限制。实际上，我们认为在不同计算机系统层次上建立应用程序的性能模型是最为重要的工作，也是高性能计算领域中的引导性原理。

第 4 章介绍共享存储和分布式存储两种类型的并行计算机体系结构，以及相关网络拓扑结构。第 5 章在理论层次上介绍并行计算概念，首先介绍一些重要的并行编程模式，然后介绍能够解释并行可扩展性限制的性能模型，因此回答了利用低速处理器构建大规模并行计算机系统的原因和方式。第 6 章对 OpenMP 进行了介绍，OpenMP 是在共享存储系统上编写

科学计算应用的主流语言。第 7 章介绍了几种典型的与 OpenMP 相关的程序性能问题并给出了相应的避免或减轻问题的方法。由于缓存一致的非一致内存访问（ccNUMA）系统已经大规模应用在高性能计算的市场中（这也是被忽视的事实，甚至一些高性能计算专家也将其忽视），相应的优化技术在第 8 章介绍。第 9 章和第 10 章介绍分布式系统上的消息传递接口语言（Message Passing Interface，MPI）以及如何编写高效 MPI 代码。最后，第 11 章介绍 MPI 和 OpenMP 混合编程方法。每章最后都配备相应的习题，我们推荐读者认真研究，这些习题包括一些书中未介绍的零碎知识或者一些特定的主题。附录 B 提供了相应的答案。

我们建议读者从头到尾认真阅读本书，因为本书的主题都极为重要。但是只想了解 OpenMP 和 MPI 的读者可以只从第 6 章和第 9 章开始阅读，并从第 7 章、第 8 章和第 10 章中学习相应的优化技术。本书自引用比较多，所以如果忽略了某些部分，也可以在其他地方找到相关材料。

致谢

本书起源于为 Springer 出版社“Lecture Notes in Physics”卷编写的两章内容，也即 2006 年计算多粒子物理夏季讨论班的内容 [A79]。我们感谢讨论班的组织者，特别是 Holger Fehske、Ralf Schneider 和 Alexander Weisse，使得我们有机会将自己在高性能计算领域的经验总结成文，虽然本书增加了很多其他材料，但是没有最初的两章内容我们就无法开始本书的撰写。

本书包括与用户、学生、算法、代码和工具一起工作的十几年经验，因此许多人都直接或者间接（甚至没有意识到）为本书做出了贡献。我们特别要感谢 Erlangen Regional Computing Center (RRZE) 高性能计算服务中心的员工，特别是 Thomas Zeiser、Jan Treibig、Michael Meier、Markus Wittmann、Johannes Habich、Gerald Schubert 和 Holger Stengel，在与他们的讨论中得到了十分有用的知识。在十年中我们小组从“Competence Network for Scientific High Performance Computing in Bavaria”(KONWIHR) 和弗里德里希 – 亚历山大 – 埃尔兰根 – 纽伦堡大学获得了极大的资助。

我们还要感谢 Uwe Küster（HLRS Stuttgart）、Matthias Müller（ZIH Dresden）、Reinhold Bader 和 Matthias Brehm（LRZ München），感谢他们在服务中心与我们的高效合作。还要特别感谢 Darren Kerbyson（PNNL）的鼓励和对我们工作的建议。最后我们要感谢 Rolf Rabenseifner (HLRS) 和 Gabriele Jost (TACC) 与我们在混合编程方面的合作，本书第 11 章就是我们一起合作的成果。

一些公司的第一手技术支持及一些合作（即使没有利润）也值得我们感谢，Intel (Andrey Semin 和 Herbert Cornelius)、SGI (Reiner Vogelsang 和 Rüdiger Wolff)、NEC (Thomas Schönemeyer)、Sun Microsystems (Rick Hetherington、Ram Kunda 和 Constantin Gonzalez)、IBM (Klaus Gottschalk) 以及 Cray (Wilfried Oed)。

我们还要感谢 CRC 员工的支持，特别是 Ari Silver、Karen Simon、Katy Smith 和 Kevin Craig。最后，如果没有 Horst Simon (LBNL/NERSC) 和 Randi Cohen（Taylor & Francis）的鼓励，本书将不可能顺利完成，他们使我们有动力开始撰写本书。

Georg Hager 与 Gerhard Wellein
埃尔兰根计算中心
埃尔兰根 – 纽伦堡大学
德国

ASCII（American Standard Code For Information Interchange） 美国标准信息交换代码

ASIC（Application-Specific Integrated Circuit） 专用集成电路

BIOS（Basic Input/Output System） 基本输入/输出系统

BLAS（Basic Linear Algebra Subroutines） 基本线性代数子程序

CAF（Co-Array Fortran） co-array Fortran

ccNUMA（cache-coherent NonUniform Memory Access） 缓存一致的非一致内存访问

CFD（Computational Fluid Dynamics） 计算流体力学

CISC（Complex Instruction Set Computer） 复杂指令集计算机

CL（Cache Line） 高速缓存行

CPI（Cycles Per Instruction） 指令周期

CPU（Central Processing Unit） 中央处理器

CRS（Compressed Row Storage） 压缩行存储

DDR（Double Data Rate） 双倍速率

DMA（Direct Memory Access） 直接内存访问

DP（Double Precision） 双精度

DRAM（Dynamic Random Access Memory） 动态随机存取内存

ED（Exact Diagonalization） 精确对角化

EPIC（Explicitly Parallel Instruction Computing） 显式并行指令计算

Flop（Floating-point operation） 浮点运算

FMA（Fused Multiply-Add） 混合乘加

FP（Floating Point） 浮点

FPGA（Field-Programmable Gate Array） 现场可编程逻辑门阵列

FS（File System） 文件系统

FSB（FrontSide Bus） 前端总线

GCC（GNU Compiler Collection） GNU 编译器集合

GE（Gigabit Ethernet） 千兆以太网

GigE（Gigabit Ethernet） 千兆以太网

GNU（GNU is not UNIX） GNU

GPU（Graphics Processing Unit） 图形处理单元

GUI（Graphical User Interface） 图形用户界面

HPC（High Performance Computing） 高性能计算

HPF（High Performance Fortran） 高性能 Fortran 语言

HT（HyperTransport） 超传输

IB（InfiniBand） infiniband

ILP（Instruction-Level Parallelism） 指令级并行

IMB（Intel Mpi Benchmarks） 英特尔 MPI 基准测试

I/O（Input/Output） 输入/输出

IP（Internet Protocol） 互联网协议

JDS（Jagged Diagonals Storage） 锯齿形对角线存储格式

L1D（Level 1 Data cache） 1 级数据缓存

L1I（Level 1 Instruction cache） 1 级指令缓存

L2（Level 2 cache） 2 级缓存

L3（Level 3 cache） 3 级缓存

LD（Locality Domain） 局部域

LD（Load） 加载

LIKWID（Like I Knew What I am Doing） 就像我知道我在做什么

LRU（Least Recently Used ） 最近最少使用策略

LUP（Lattice site UPdate） 网格更新

MC（Monte Carlo） 蒙特卡洛方法

MESI（Modified/Exclusive/Shared/Invalid） 修改/专有/共享/无效（缓存一致性协议状态）

MI（Memory Interface） 存储器接口

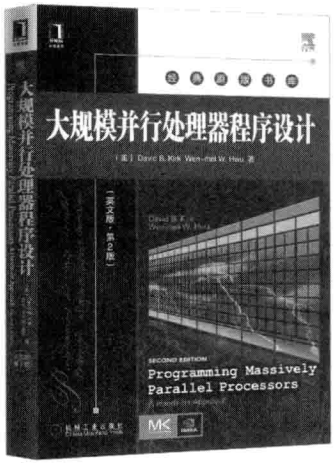
MIMD（Multiple Instruction Multiple Data） 多指令流多数据流

MIPS（Million Instructions Per Second） 每秒百万指令

MMM（Matrix-Matrix Multiplication） 矩阵矩阵乘

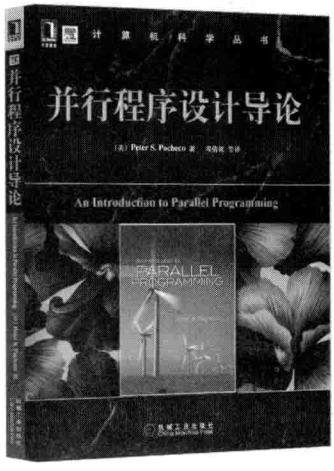
MPI（Message Passing Interface） 消息传递接口
MPMD（MultiPle Program Multiple Data） 多程序流多数据流
MPP（Massively Parallel Processing ） 大规模并行处理
MVM（Matrix-Vector Multiplication） 矩阵向量乘
NORMA（NO Remote Memory Access） 非远程存储访问
NRU（Not Recently Used） 最近未使用算法
NUMA（NonUniform Memory Access） 非一致内存访问
OLC（Outer-Level Cache） 外级缓存
OS（Operating System） 操作系统
PAPI（Performance Application Programming Interface） 应用程序性能编程接口
PC（Personal Computer） 个人电脑
PCI（Peripheral Component Interconnect） 外围组件互连
PDE（Partial Differential Equation） 偏微分方程
PGAS（Partitioned Global Address Space ） 分割全局地址空间
PLPA（Portable Linux Processor Affinity） 可移植Linux 处理器亲缘性
POSIX（Portable Operating System Interface for uniX） 可移植操作系统接口（UNIX）
PPP（Pipeline Parallel Processing） 流水线并行处理
PVM（Parallel Virtual Machine） 并行虚拟机
QDR（Quad Data Rate） 4 倍数据倍率
QPI（Quick Path Interconnect） 快速路径互连
RAM（Random Access Memory） 随机访问内存
RISC（Reduced Instruction Set Computer） 精简指令集计算机
RHS（Right Hand Side） 右端项
RFO（Read For Ownership） 所有者处理
SDR（Single Data Rate） 单倍数据速率
SIMD（Single Instruction Multiple Data） 单指令流多数据流
SISD（Single Instruction Single Data） 单指令流单数据流
SMP（Symmetric MultiProcessing） 对称多处理
SMT（Simultaneous MultiThreading） 同步多线程
SP（Single Precision） 单精度
SPMD（Single Program Multiple Data） 单程序流多数据流
SSE（Streaming SIMD Extensions） SIMD 流指令扩展
ST（STore） 存储
STL（Standard Template Library） 标准模板库
SYSV（UNIX System V） UNIX 系统 V
TBB（Threading Building Blocks） TBB 语言
TCP（Transmission Control Protocol） 传输控制协议
TLB（Translation Lookaside Buffer） 旁路缓冲器
UMA（Uniform Memory Access） 一致内存访问
UPC（Unified Parallel C） UPC 语言

推荐阅读



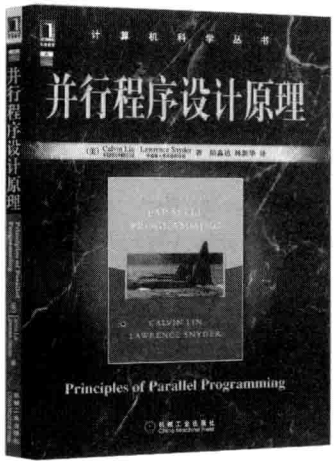
大规模并行处理器程序设计（英文版·第2版）

作者：David B. Kirk 等 ISBN：978-7-111-41629-6 定价：79.00元



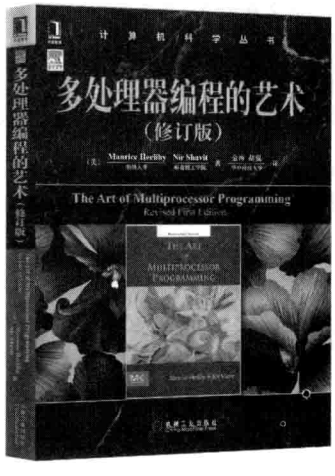
并行程序设计导论

作者：Peter S. Pacheco ISBN：978-7-111-39284-2 定价：49.00元



并行程序设计原理

作者：Larry Snyder 等 ISBN：978-7-111-27075-1 定价：45.00元



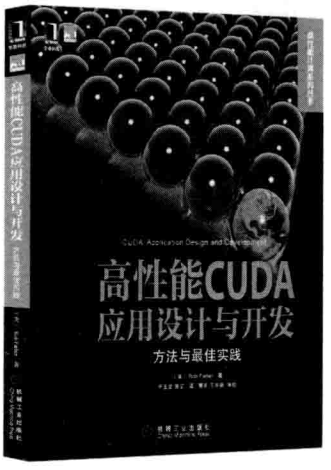
多处理器编程的艺术（修订版）

作者：Maurice Herlihy 等 ISBN：978-7-111-41858-0 定价：69.00元

推荐阅读

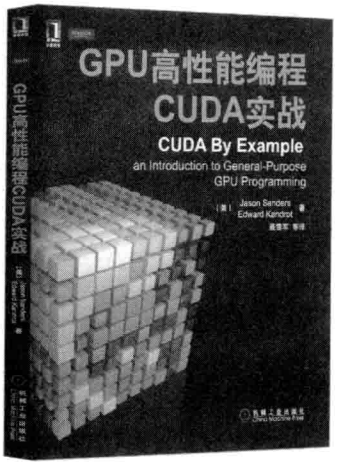
CUDA并行程序设计：GPU编程指南

作者：Shane Cook ISBN：978-7-111-44861-7 定价：99.00元



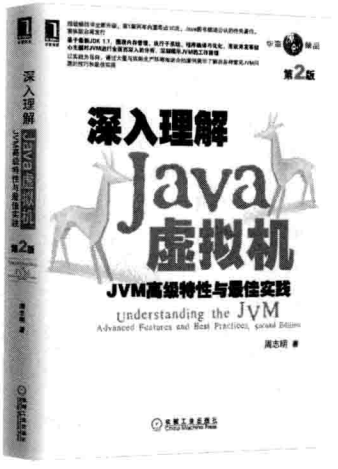
高性能CUDA应用设计与开发：方法与最佳实践

作者：Rob Farber ISBN：978-7-111-40446-0 定价：59.00元



GPU高性能编程CUDA实战

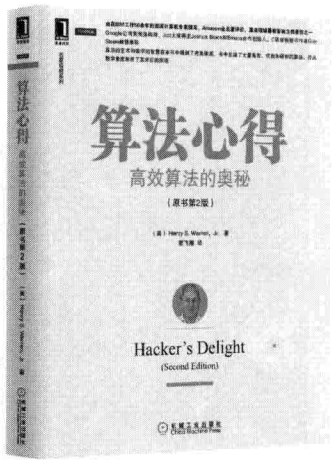
作者：Jason Sanders 等 ISBN：978-7-111-32679-3 定价：39.00元



深入理解Java虚拟机：JVM高级特性与最佳实践（第2版）

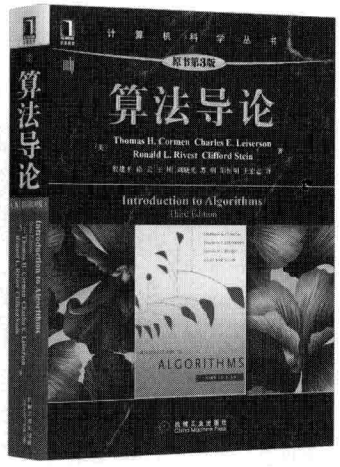
作者：周志明 ISBN：978-7-111-42190-0 定价：79.00元

推荐阅读



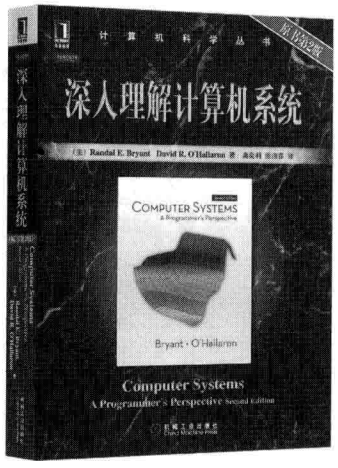
算法心得：高效算法的奥秘（原书第2版）

作者：Henry S. Warren ISBN：978-7-111-45356-7 定价：89.00元



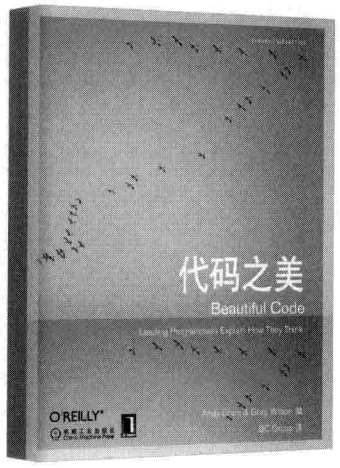
算法导论（原书第3版）

作者：Thomas H. Cormen 等 ISBN：978-7-111-40701-0 定价：128.00元



深入理解计算机系统（原书第2版）

作者：Randal E. Bryant 等 ISBN：978-7-111-32133-0 定价：99.00元



代码之美

作者：Grey Wilson ISBN：978-7-111-25133-0 定价：99.00元

出版者的话
译者序
序
前言
缩略词

第1章　当代处理器 …… 1
1.1　存储程序的计算机体系结构 …… 1
1.2　基于高速缓存的通用微处理器体系结构 …… 2
1.2.1　性能指标和基准测试 …… 2
1.2.2　晶体管：摩尔定律 …… 5
1.2.3　流水线 …… 6
1.2.4　超标量 …… 10
1.2.5　SIMD …… 10
1.3　存储层次 …… 11
1.3.1　高速缓存 …… 11
1.3.2　高速缓存映射 …… 13
1.3.3　预取 …… 15
1.4　多核处理器 …… 17
1.5　多线程处理器 …… 19
1.6　向量处理器 …… 20
1.6.1　设计原理 …… 21
1.6.2　最高性能估计 …… 22
1.6.3　程序设计 …… 23
习题 …… 25
第2章　串行代码基本优化技术 …… 26
2.1　标量剖析 …… 26
2.1.1　基于函数和代码行的程序剖析 …… 26
2.1.2　硬件性能计数器 …… 29
2.1.3　手工代码插入 …… 32
2.2　优化常识 …… 32
2.2.1　少做工作 …… 32
2.2.2　避免耗时运算 …… 32
2.2.3　缩减工作集 …… 33
2.3　小方法，大改进 …… 33
2.3.1　消除常用子表达式 …… 33
2.3.2　避免分支 …… 34
2.3.3　使用SIMD指令集 …… 34
2.4　编译器作用 …… 36
2.4.1　通用优化选项 …… 37
2.4.2　内联 …… 37
2.4.3　别名 …… 37
2.4.4　计算准确性 …… 38
2.4.5　寄存器优化 …… 39
2.4.6　利用编译日志 …… 39
2.5　C++优化 …… 40
2.5.1　临时变量 …… 40
2.5.2　动态内存管理 …… 42
2.5.3　循环与迭代器 …… 43
习题 …… 43
第3章　数据访存优化 …… 45
3.1　平衡分析与lightspeed评估 …… 45
3.1.1　基于带宽的性能建模 …… 45
3.1.2　STREAM 基准测试 …… 47
3.2　存储顺序 …… 49
3.3　案例分析：Jacobi算法 …… 50
3.4　案例分析：稠密矩阵转置 …… 53
3.5　算法分类和访存优化 …… 56
3.5.1　$O(N)/O(N)$ …… 56
3.5.2　$O(N^2)/O(N^2)$ …… 57
3.5.3　$O(N^3)/O(N^2)$ …… 60
3.6　案例分析：稀疏矩阵向量乘 …… 61
3.6.1　稀疏矩阵的存储机制 …… 62
3.6.2　JDS sMVM优化 …… 64

习题 …… 66
第4章 并行计算机 …… 68
4.1 并行计算模式分类 …… 69
4.2 共享存储计算机 …… 69
4.2.1 cache一致性 …… 69
4.2.2 UMA …… 71
4.2.3 ccNUMA …… 71
4.3 分布式存储计算机 …… 73
4.4 混合型系统 …… 74
4.5 网络 …… 75
4.5.1 网络的基本性能特征 …… 75
4.5.2 总线 …… 78
4.5.3 交换网络和胖树网络 …… 79
4.5.4 Mesh 网络 …… 81
4.5.5 混合网络 …… 82
习题 …… 82
第5章 并行性基础 …… 83
5.1 为什么并行化 …… 83
5.2 并行性 …… 83
5.2.1 数据并行性 …… 84
5.2.2 功能并行性 …… 86
5.3 并行扩展性 …… 87
5.3.1 限制并行执行的因素 …… 87
5.3.2 可扩展性指标 …… 88
5.3.3 简单可扩展性定律 …… 89
5.3.4 并行效率 …… 90
5.3.5 串行性能与强可扩展性 …… 91
5.3.6 改进的性能模型 …… 92
5.3.7 选择正确的扩展性基准 …… 94
5.3.8 案例分析：低速处理器计算机能否变得更快 …… 95
5.3.9 负载不均衡 …… 98
习题 …… 101
第6章 使用OpenMP进行共享存储并行编程 …… 103
6.1 OpenMP简介 …… 103
6.1.1 并行执行 …… 103
6.1.2 数据作用域 …… 105
6.1.3 循环的OpenMP工作共享 …… 106
6.1.4 同步 …… 107
6.1.5 归约 …… 108
6.1.6 循环调度 …… 109
6.1.7 任务 …… 110
6.1.8 其他方面 …… 111
6.2 案例分析：OpenMP并行实现Jacobi算法 …… 112
6.3 高级OpenMP：波前并行化 …… 114
习题 …… 116
第7章 高效OpenMP编程 …… 119
7.1 OpenMP程序性能分析 …… 119
7.2 性能缺陷 …… 120
7.2.1 减轻Open MP共享区开销 …… 121
7.2.2 决定短循环的OpenMP开销 …… 126
7.2.3 串行化 …… 128
7.2.4 伪共享 …… 129
7.3 案例分析：并行稀疏矩阵向量乘 …… 130
习题 …… 133
第8章 ccNUMA体系结构的局部性优化 …… 134
8.1 ccNUMA的局部访问 …… 134
8.1.1 首次访问方式分配页面 …… 135
8.1.2 通过其他方式的局部性访问 …… 137
8.2 案例分析：稀疏MVM的ccNUMA优化 …… 138
8.3 页面布局缺陷 …… 139
8.3.1 非NUMA友好的OpenMP调度 …… 139
8.3.2 文件系统高速缓存 …… 140
8.4 C++中的ccNUMA问题 …… 142
8.4.1 对象数组 …… 142
8.4.2 标准模板库 …… 144
习题 …… 146
第9章 使用MPI进行分布式存储并行内存编程 …… 147
9.1 消息传递 …… 147
9.2 MPI简介 …… 148

9.2.1 一个简单例子 …… 148
9.2.2 消息和点对点通信 …… 150
9.2.3 集合通信 …… 154
9.2.4 非阻塞点对点通信 …… 157
9.2.5 虚拟拓扑 …… 160
9.3 实例：Jacobi解法器的MPI并行 …… 162
9.3.1 MPI实现 …… 162
9.3.2 性能特征 …… 167
习题 …… 170
第10章 高效MPI编程 …… 171
10.1 MPI性能工具 …… 171
10.2 通信参数 …… 174
10.3 同步、串行化和竞争 …… 174
10.3.1 隐式串行化和同步 …… 174
10.3.2 竞争 …… 176
10.4 降低通信开销 …… 177
10.4.1 最优化区域分解 …… 177
10.4.2 聚合消息 …… 180
10.4.3 非阻塞与异步通信 …… 181
10.4.4 集合通信 …… 183
10.5 理解节点内点对点通信 …… 184
习题 …… 189
第11章 MPI与OpenMP混合编程 …… 190
11.1 基本MPI/OpenMP混合编程模型 …… 190
11.1.1 向量模式实现 …… 191
11.1.2 任务模式实现 …… 191
11.1.3 案例分析：混合Jacobi解法器 …… 192
11.2 MPI线程交互分类 …… 193
11.3 混合分解及映射 …… 195
11.3.1 每个节点一个MPI进程 …… 195
11.3.2 每个插槽一个MPI进程 …… 196
11.3.3 每个插槽多个MPI进程 …… 196
11.4 混合编程的优势和劣势 …… 197
11.4.1 改善的收敛速度 …… 197
11.4.2 共享高速缓存中的数据重用 …… 197
11.4.3 利用额外级别的并行性 …… 198
11.4.4 重叠MPI通信和计算 …… 198
11.4.5 减少MPI开销 …… 198
11.4.6 多级别开销 …… 198
11.4.7 向量模式下批量同步通信 …… 198
附录A 多核环境中的拓扑和亲缘性 …… 199
A.1 拓扑 …… 200
A.2 线程和进程分布 …… 201
A.2.1 外部亲缘性工具 …… 201
A.2.2 程序控制亲缘性 …… 203
A.3 非页面首次访问分配策略 …… 204
附录B 习题解答 …… 206
参考文献 …… 221
索引 …… 232

当代处理器

在 1975 ～ 1995 年的“旧时代”的科学计算时期，先进的高性能系统是专门为 HPC 市场设计的，主要的厂商有 Cray、CDC、NEC、Fujitsu 和 Thinking Machines 等。在性能和价格方面，这些系统远远超越了标准的“商品”电脑。20 世纪 70 年代初发明的单芯片通用微处理器，是 20 世纪 80 年代末唯一足够成熟、可以打入 HPC 市场的技术。直到 20 世纪 90 年代末，标准的工作站集群甚至基于 PC 的硬件至少在理论峰值性能上才具备相应的竞争力。如今，情况已经发生了很大变化。HPC 世界被低成本、现成的处理器系统占领，这些系统并不是主要为科学计算而设计的。一些传统的超级计算机厂商在这个有利可图的市场上活动，他们提供在单 CPU 水平上的高应用性能以及高度并行工作负载的系统。因此，科学家和工程师很可能遇到这样的“商品模式”，即随着需求的增长推动硬件向更为专业的方向发展。因此，本章将主要关注基于标准的高速缓存微处理器的系统。向量机（vector computer）支持不同的编程方式，这种编程方式在很多方面更接近科学计算的要求，但是目前向量机已非常罕见。然而如果没有它们，对超级计算机体系结构的讨论将不完整，1.6 节对此提供了相关的概述。

1.1 存储程序的计算机体系结构

当讨论计算机系统时，我们的脑海中总是有一个清楚的体系结构概念，即 1936 年由图灵提出并由 Eckert 和 Mauchly 实现的第一个真正的计算机 EDVAC [C129,C131]。图 1-1 显示了存储程序数字计算机的简图，将指令作为数据存储在内存，这是它不同于早期设计的最大特征。指令由控制单元读取并执行，一个独立的算术逻辑单元根据指令来负责实际的计算并操作内存中的数据。通过 I/O 设备可以与用户通信。中央处理单元（CPU）包括控制单元、算术逻辑单元、适合的内存接口和 I/O 接口。在存储程序计算机上编程意味着修改内存中的指令，原则上这可以由另一个程序完成。编译器就是一个典型的例子，因为它将 C 或 Fortran 等高级语言翻译成能够在内存中存储并能被计算机执行的指令。

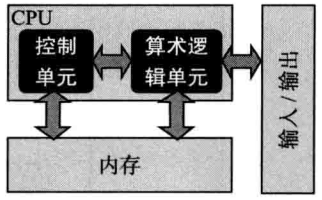


图 1-1 存储程序计算机体系结构概念。将程序指令作为数据存储在内存

这个简图是现今所有主流的计算机系统的基础，但仍然存在一些固有问题：

- 必须连续不断地向控制和算术逻辑单元提供数据，因此内存接口速度成为计算性能的瓶颈，这通常称为冯·诺依曼瓶颈。在接下来的章节中，我们将展示如何利用体系结构优化和编程技术减弱这个瓶颈导致的不利影响。尽管如此，这个瓶颈仍然是最严重的性能限制因素。
- 这种体系结构存在内在顺序性，一条指令可能处理来自内存的一个或一组操作数。

SISD（单指令单数据，Single Instruction Single Data）即为这种概念。如何修改和扩展使得它能够以多种不同方式支持并行，以及这样一个并行机如何能够有效运用，同样是这本书的主题。

尽管存在着这些缺陷，但没有其他的体系结构概念能在将近 70 年的电子数字计算机中如此广泛地使用。

1.2 基于高速缓存的通用微处理器体系结构

微处理器可能是人类发明的最复杂的机器。然而，就像前面章节中描述的那样，它们都基于存储程序数字计算机的概念。对于科学家而言，理解 CPU 所有内部工作细节是不可能的，也是不必要的，尽管把握其高级特性对于了解潜在的性能瓶颈是有帮助的。图 1-2 展示了现代基于高速缓存的通用微处理器的简图。对于一个运行的程序，真正执行计算的部分

是仅占芯片一小部分的浮点型（FP）和整型（INT）计算单元。其他逻辑控制单元用来向计算单元提供操作数。一般将 CPU 寄存器区分为浮点数和整数两种类型，CPU 寄存器能够存放指令访问所需操作数，这样在数据访问时没有明显的延时。在一些体系结构中，所有算术运算的操作数都必须存放在寄存器中。如今，主流 CPU 有 16 到 128 个两种类型的寄存器。Store（ST）和 Load（LD）单元执行寄存器存取指令，待执行指令被保存到多个队列中，这些指令可能会被乱序执行。最后，高速缓存保存即将执行的指令和数据。而芯片的主要部分就是高速缓存。

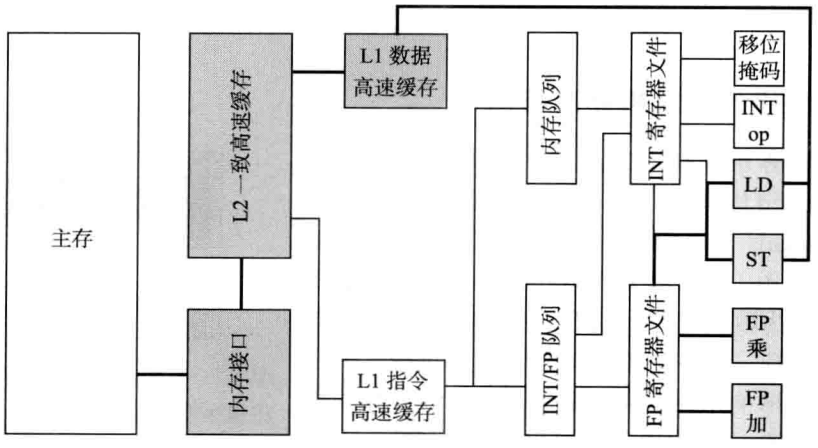


图 1-2 典型的基于高速缓存的微处理器（单核）的简单框图。同一芯片上的其他核能够共享高速缓存和内存接口等资源。与科学计算性能相关的功能模块和数据通路在图中突出显示了

许多额外逻辑，包括分支预测、缓冲区重排、数据旁路、事务队列等，已经应用到现代微处理器中，在此我们不进行讨论。供应商提供了许多文档描述这些技术细节[V104,V105,V106]。在过去十年中，多核处理器已经取代了传统单核处理器设计模式。在多核芯片上，多个处理器（“核”）同时执行代码。它们能够像存储器接口或高速缓存一样，在不同程度上共享资源，详细内容见 1.4 节。

1.2.1 性能指标和基准测试

峰值性能是 CPU 核心所有组件同时执行时的最高性能。一个特定的应用程序能否达到

峰值取决于许多因素（详见第 3 章内容）。这里，我们介绍一些衡量 CPU 利用率的基本性能指标。科学计算通常使用“双精度”（DP）浮点数运算，FP 单元的性能测量标准是每秒执行多少次浮点乘法和加法运算（Flop/s，浮点数运算次数 / 秒）。由于更复杂运算（除法、平方根、三角函数等）与乘法及加法共享计算资源，并且执行效率很低，对于实际性能衡量没有意义（见第 2 章），因此高性能软件应该尽量避免调用这样的操作。在本书写作时，标准商用微处理器通常在每个时钟周期最多完成两个或四个双精度浮点计算。在时钟频率为 2 ～ 3 GHz 时，峰值为每核 4 ～ 12GFlop/s。 3

如上所述，向运算单元提供操作数是一项复杂的工作。从程序员的角度看，最重要的数据通路是从高速缓存到主存。性能指标用数据带宽即 GB/s 来衡量。GFlop/s 和 GB/s 通常是衡量微处理器性能的主要指标[⊖]。因此，如图 1-3 所示，从一个关注性能的程序员角度看，一个基于高速缓存的微处理器是以数据为中心的，计算或某种算法通常由数据项的操作决定。然而，算法的具体实现受到硬件数据通路的性能限制，特别是主存。

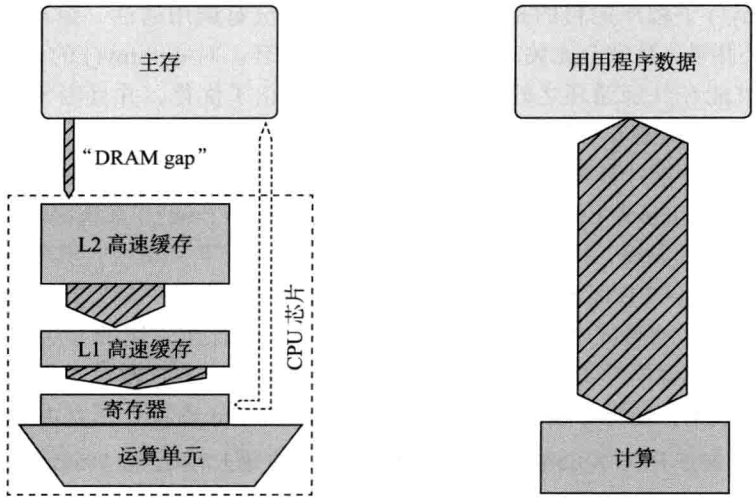


图 1-3 左图是一个基于高速缓存的微处理器的存储层次结构简图（在所有的体系结构中不能直接从寄存器到内存进行数据传输），通常有一个单独的 L1 高速缓存存储指令。右图中的“ DRAM gap”展示了主存与高速缓存之间带宽的巨大差异。这个模型必须映射为具体应用的访存模式

通常用低级别的基准测试衡量处理器或者系统的性能特征，即只测量系统中某些特征的程序，比如峰值性能或存储带宽等。一个突出的例子是由 Schönauer 提出的三维向量算法 [S5]。它包括一个嵌套循环，循环内执行一个三维向量的复合乘法，并将结果保存在第四个向量中（见代码清单 1-1 中的 10 ～ 15 行）。该基准测试的目的是衡量处理器的内存和运算单元之间的数据传输性能。在内循环中，3 个 load 流 B、C 和 D 以及一个 store 流 A 是活跃的，根据 N 的不同，该循环的时间可能非常短而难以衡量。外部循环再重复 R 次，以使执行时间变得足够长而能被精确测量。实际中，人们往往会根据 N 而选择 R，从而使得总体的执行时间保持不变。 4

⊖ 请注意，当与带宽和运算性能等一起用时，G、M 分别表示 10^9 和 10^6，但是一般情况下 G、M 表示 2 的多少次方，例如 1MB = 2^{20} 字节。

代码清单 1-1 三维向量算法基准测试的简单代码，包括性能评估

```
double precision, dimension(N) :: A,B,C,D
double precision :: S,E,MFLOPS

do i=1,N                              !initialize arrays
  A(i) = 0.d0; B(i) = 1.d0
  C(i) = 2.d0; D(i) = 3.d0
enddo

call get_walltime(S)                  ! get time stamp
do j=1,R
  do i=1,N
    A(i) = B(i) + C(i) * D(i)         ! 3 loads, 1 store
  enddo
  if(A(2).lt.0) call dummy(A,B,C,D) ! prevent loop interchange
enddo
call get_walltime(E)                  ! get time stamp
MFLOPS = R*N*2.d0/((E-S)*1.d6)        ! compute MFlop/sec rate
```

调用 dummy() 子程序的目的是阻止编译器优化。没有调用的话，编译器会发现，内层循环和外层循环指数 j 是完全无关的，从而删除外层循环。对 dummy() 的调用使得编译器认为，这些数组可能在外层循环之间变换，这有效地防止了优化，并且因为 if 语句条件总是非真（这个编译器不知道），所以调用 dummy() 的开销可以忽略不计。

MFLOPS 变量是指计算整个循环嵌套的 MFlop/s。请注意，在基准测试中最合理的时间单位是墙上时间 (wallclock time)，也称为逝去时间 (elapsed time)。系统提供的任何其他计量时间方式很容易产生误解，因为可能计量了包含 I / O、上下文切换、其他进程开销等的时
5 间，这不应该计算在 CPU 时间之中。对于并行程序时间的计量更是如此（见第 5 章）。像上面提到的三维向量基准测试中用到的时间计量方法是一个比较常用的方法，如代码清单 1-2 所示。提供具有和不具有强调功能的两个版本，是因为 Fortran 常常对子程序名称给予强调。有了这两个有效版本，使得将编译过的 C 代码链接到 Fortran 或者 C 语言主程序上总是可行。

代码清单 1-2 基于 POSIX gettimeofday() 功能的 C 程序墙上时间。在 Windows 操作系统中，GetSystemTimeAsFileTime() 函数具有相同的功能

```
#include <sys/time.h>

void get_walltime_(double* wcTime) {
  struct timeval tp;
  gettimeofday(&tp, NULL);
  *wcTime = (double)(tp.tv_sec + tp.tv_usec/1000000.0);
}

void get_walltime(double* wcTime) {
  get_walltime_(wcTime);
}
```

图 1-4 显示了三维向量算法的测试程序在向量系统和一些不同的基于高速缓存的微处理器上的性能。在非常小的循环上，无论哪种类型的 CPU 或体系结构计算性能都很差，但是随着 N 的增大，标准微处理器的计算性能一直增加到最高性能，随后骤然下降。最后，对于非常大的循环，性能保持恒定。这些特性将在 1.3 节详细分析。

6 向量处理器（图 1-4 中的虚线）显示出突出的特征。它的低性能区域延伸得比基于高速缓存的微处理器更远，但是它完全没有等于 0。我们得出这样的结论：向量系统对于标准的 CPU 具有互补性，因为不同的 CPU 用在不同领域（1.6 节详细介绍向量处理器体系结构）。

然而在实际的代码优化中，也许能够通过避免低性能区域来提高性能（详细内容介绍见第 2 章和第 3 章）。

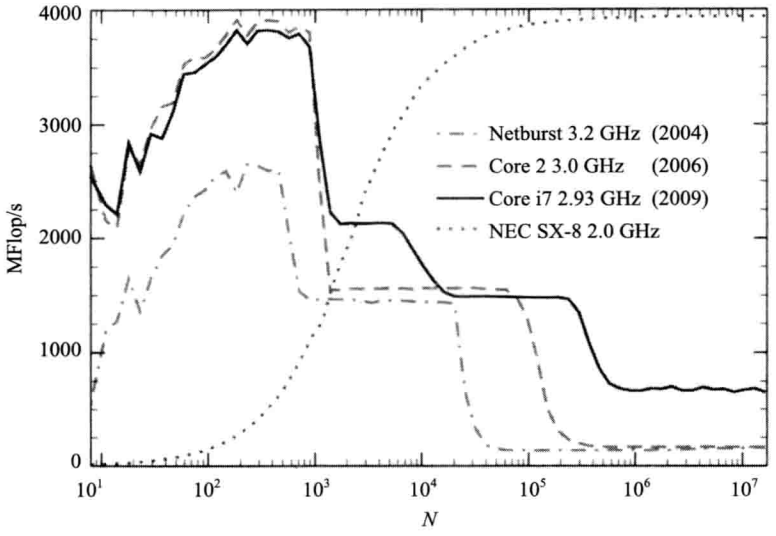


图 1-4 一些英特尔系列处理器（时钟频率和生产年份在图中已经标出）和 NEC SX-8 向量处理器对串行三维向量算法的性能与循环长度。后续章节将解释不同性能的详细特点

低层次的基准测试是获得处理器基本性能信息的有力工具。然而它们不能够准确地预测出每一个实际应用程序的运行情况。为了确定某些 CPU 或体系结构是否能够很好地适应一些应用程序（例如，在采购前或在为一个计算机时间写建议前），唯一安全的方法是制定应用基准测试。这意味着，应用程序代码运行时会需要一些输入参数，这些参数能真实地反映实际运行的要求。应该基于大量应用基准测试决定支持或反对一个体系结构。像 SPEC 这种标准的测试基准只能作为一个粗糙的指南 [W118]。

1.2.2 晶体管：摩尔定律

台式机出现之前的计算技术已经被广泛用于具有大量计算需求的科学计算中，在超过 30 年的时间里，无论哪种技术构造的计算机芯片，它们的复杂性或通用能力每 24 个月大约翻一倍，这一趋势称为摩尔定律。英特尔公司创始人之一 Gordon Moore，在 1965 年预测一个芯片上可以容纳的最小晶体管数量将会按照预测的速率增长 [R35]。尽管制造技术已经发生了质的变化，但这一规律从 20 世纪 60 年代早期一直持续到现在。令人惊讶的是，处理器并不是计算机的唯一组成部分，所以处理器性能的意义仍然存在很大争议，但处理器芯片复杂性的增长总是大约等同于计算机性能的增长（下文对这点有更多讨论）。

通过增加芯片中晶体管的数量和时钟频率，处理器的设计者能够实现许多先进的技术，从而使计算机应用性能得到改善。于是产生了许多新的概念：

1）**流水线功能单元**。在所有关于计算机技术的创新中，流水线也许是最重要的一个。通过将复杂的操作（例如，浮点数加法和乘法）分解成能够在 CPU 上不同的功能单元执行的简单部分，可以提高指令吞吐量，即增加每个时钟周期执行的指令条数。这是指令级并行（Instruction Level Parallelism，ILP）最基本的例子。最佳的流水线能够得到每个时钟周期一

7 条指令的吞吐量。在本书写作时，处理器的流水线存在30多个阶段。详细介绍见1.2.3节。

2）**超标量体系结构**。通过使得每个时钟周期中指令的吞吐量大于1，超标量结构直接提供了指令级并行。这需要更多相同的功能单元，使得操作能够同时执行（详细介绍见1.2.4节）。现代微处理器实现了六路超标量体系结构。

3）**通过单指令多数据（SIMD）指令达到数据并行**。通常针对特殊寄存器上的整型或浮点型向量数组，SIMD（Single Instruction Multiple Data）指令能发出相同的操作。这在不添加超标量技术的情况下也能提升算法性能。例如，英特尔SSE系列机，AMD的3DNow！基于Power和PowerPC处理器扩展的AltiVec，以及Sun公司的UltraSPARC设计中的“VIS”指令集（详细介绍见1.2.5节）。

4）**乱序执行**。如果指令的操作数不能及时从寄存器中获得，例如存储系统太慢跟不上处理器的速度，此时乱序执行则可以允许执行后续指令流中已经获得参数的指令，从而避免阻塞时间（也叫做stall）。这能提高指令吞吐量，同时使得编译器的代码组织和性能优化更加简单。通过使用重排序缓冲区来存储待执行的指令直到该指令适合执行为止，目前的乱序执行设计在任何时候都能够保持数百条指令的执行速率。

5）**更大的高速缓存**。由于处理器和存储器的速度差距越来越大（见1.3节），小型、快速的片上存储器用来暂存数据副本，这些刚使用过的数据或者位置靠近它们的数据可能很快被再次使用。增大高速缓存的大小并不影响应用程序性能，但是仍然需要一个折中，因为大型高速缓存比小型的速度要慢。

6）**精简指令集**。在20世纪80年代，指令集从CISC（Complex Instruction Set Computer，复杂指令集）转变到RISC（Reduced Instruction Set Computer，精简指令集）。在CISC中，处理器执行的指令非常复杂、功能强大，需要大型硬件对指令解码，使程序简短精湛。这减轻了程序员的负担，同时节约了稀缺的内存资源。RISC的特征是拥有一组非常简单的指令集，能够使程序的执行非常迅速（每条指令只要几个时钟周期，在极端情况下能到达每条指令仅需一个时钟周期）。RISC微处理器频率的增长速率要远远超过CISC的增长速率，此外它能够节省出更多晶体管以做他用。如今大部分用于科学计算的计算机系统在低层都使用RISC。而基于x86的处理器虽然执行CISC的机器代码，但它们却在内部将其转化为RISC模式执行。

8 尽管有这么多创新，但最近处理器厂商一直面临着很大障碍，很难再将单核CPU的极限性能推向一个新的层次。摩尔定律揭示了晶体管数量的稳定增长，但是更加复杂并不等价于更加有效。相反，越多的功能单元被设计到CPU中，代码不使用它们的可能性也就越大。因为在一个指令流中，独立指令的数量是有限的。此外，根据摩尔定律，时钟频率的稳定增长需要与单核性能保持同步，然而更快的时钟会增加功耗，从而使得空转晶体管更加无用。

在探索这个性能瓶颈解决方法时，人们尝试过通过放弃一些系统结构复杂性以更加直接的思路来简化处理器的设计。使用额外的晶体管以获得更大的高速缓存是思路之一，但是同样存在着局限性，更大的高速缓存不会对性能有很大的提升。多核处理器，即在单个晶片或插槽上放置多个CPU核，是如今大部分制造厂商选择的解决办法。1.4节将对其做详细讲解。

1.2.3 流水线

微处理器流水线与生产装配线有着异曲同工之妙。工人（功能单元）不需要知道最终产品的所有细节，就能够熟练、专业地完成一个单一的任务。对于不同的产品，每个工人一遍

又一遍地执行相同的工作，再将半成品交给生产线上的下一个工人。如果完成一个产品需要 m 个不同的步骤，则会有 m 个不同的产品同时出现在生产线的不同阶段。如果生产线上所有阶段花费的时间相同，则所有的工人会一直处于繁忙状态。最终在每个阶段的时间内将会有一个成品产生。

诸如存取数据或浮点数运算这种复杂操作，如果没有额外的硬件支持无法在单个时间周 9
期内完成。幸运的是，装配线的概念在这里同样适用。最简单的流水线是“取指令 - 分析指令 - 执行”，这样每个阶段可以和其他阶段独立地分开。当执行一条指令时，正在分析另一条指令，而第三条指令正在从指令高速缓存（L1I）中取出。通常这些仍然复杂的任务会被进一步细分。由于功能单元变得简单，细分任务可以获得更高的时钟速率。以浮点数运算为例，如图 1-5 所示，它被分解成 5 个简单的子任务。一个向量积 A(:) = B(:)*C(:)，从第一阶段开始执行，对元素 B(1) 和 C(1) 分离尾数和指数。这时剩下的四个功能单元是空闲的。于是，中间结果被交给第二阶段，而第一阶段开始处理 B(2) 和 C(2)。在第二个周期中，有 3/5 的功能单元仍然是空闲的。在第四个时钟周期之后，流水线完成了它所谓的“排空”阶段。换句话说，多级流水线有一个五个时钟周期的延迟（或深度），过了这个时间，会产生第一个运算结果。然后，所有的功能单元会一直处于工作状态，每个时钟周期将会产生一个结果。因此，我们得到了每周期一条指令的吞吐量。当流水线的第一阶段完成了 B(N) 和 C(N) 的操作时，“排空”阶段开始了。四个周期之后循环结束，得到了所有的结果。

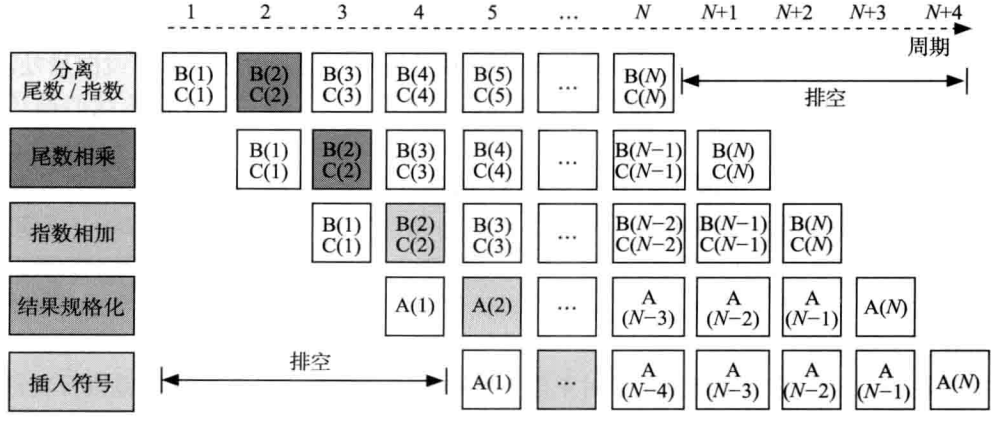


图 1-5 执行浮点型 A(:) = B(:)*C(:) 乘法流水线的时间线。在四个周期的排空时间之后，每个周期都会产生一个结果

一般情况下，对于 m 级流水线，执行 N 个独立的连续操作需要 $N + m - 1$ 步。因此，对于一个需要 m 个时钟周期产生一个结果的通用单元，我们可以预算出其加速比，

$$\frac{T_{\text{seq}}}{T_{\text{pipe}}} = \frac{mN}{N+\text{m}-1} \tag{1-1}$$

对于很大的 N，其值趋近 m，则吞吐量是：

$$\frac{N}{T_{\text{pipe}}} = \frac{1}{1+\dfrac{m-1}{N}} \tag{1-2}$$

由于 N 很大，其结果接近于 1（见图 1-6）。显然，由于排空阶段的开销，为了达到合理

的吞吐量，更深的流水线意味着独立操作的数量应该更大。

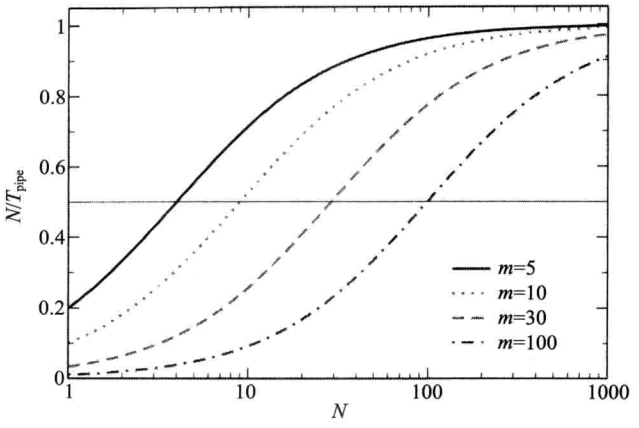


图 1-6 流水线吞吐量与独立功能部件个数的函数关系。m 是流水线的深度

容易确定，为了得到至少每周期 p 的吞吐量需要多大的 N（$0 < p \leqslant 1$）：

$$p = \frac{1}{1 + \frac{m-1}{N_c}} \Rightarrow N_c = \frac{(m-1)p}{1-p} \tag{1-3}$$

当 $p = 0.5$ 时，$N_c = m-1$。考虑到当今微处理器的整体流水线长度在 10 ～ 35 个阶段之间，我们能用短小紧凑的循环很快找出代码中潜在的性能瓶颈。对于超标量或向量处理器，情况会变得更加糟糕，因为它们使用多个相同的并行流水线，使得每个流水线的循环长度变得更短。

10 关于流水线的另一个问题是，执行诸如 FP 除法或是超越函数这样的复杂运算往往有很长的延迟（平方根和除法需要几十个周期，而三角函数需要更多甚至超过 100 个周期），它们只有很少的流水性或是根本无法流水执行，以至于整个流水线上的指令不得不被延迟，导致了所谓的流水线气泡。避免这一现象是代码优化的主要目标，这和其他相关的流水线优化一起将在第 2 章谈及。

需要注意的是，尽管五级流水线对于浮点数乘法流水线并不是不可行，但执行真正的代码时会涉及更多的操作，比如加载、存储、取值、译码及指令分析等，会与运算有很多重叠部分。每个指令的操作数必须从内存取到寄存器，每个结果必须被写回内存，还需要发现所有可能的依赖关系。为了使流水线的各个阶段更加有效，编译器需要安排指令的执行顺序。这对于顺序执行体系结构至关重要，同样对于存在某些大量延迟操作的乱序执行的处理器也很关键。

正如上面提到的，一条指令只有在其操作数有效时才能够被执行。如果操作数没有被及时送到执行单元，那么所有复杂的流水线机制都没有用。以下面的一个小规模循环作为例子：

```
1 do i=1,N
2   A(i) = s * A(i)
3 enddo
```

从高级语言的角度来看它很简单，然而这个循环对应着 RISC 处理器的大量汇编指令。对应的伪代码如下：

```
1 loop:  load A(i)
2        mult A(i) = A(i) * s
3        store A(i)
4        i = i + 1
5        branch -> loop
```

11

尽管乘法运算可以设计成流水线，但是如果 A(i) 上的加载操作不及时提供数据，流水线会被延迟。同样，只有在乘法操作被执行以及得到有效结果后，数据存储操作才能够被执行。假设数据加载操作需要 4 个时钟周期，乘法运算和数据存储均需要 2 个时钟周期，那么很明显，上面的伪代码是很低效的。这就需要插入不相关的循环指令来避免流水线的延迟和阻塞。

当然，代码中会有排空时间，在这里没有展示。我们简单地假设 CPU 能够在单个时钟内发出一个循环中的所有四条指令，并且循环变量的增加和最后一个分支操作没有任何开销。交错循环的顺序来弥补时间延迟称为软流水。这种优化需要熟悉处理器体系结构和应用程序代码的编译原理。通常为达到“最佳”代码，会应用启发式算法。

但是，并不能总是通过软流水优化指令序列。由于循环中数据依赖关系的存在（例如一个循环依赖于另一个循环的结果），编译器和处理器硬件都不能避免流水线的阻塞。对于前面例子中的小规模循环，为了计算 A(i) 需要 A(i+offset)，而 offset 是一个编译时已知的常数或者变量。

12

真相关	伪相关	通用版本
do i=2,N A(i)=s*A**(i-1)** enddo	do i=i,N-1 A(i)=s*A**(i+1)** enddo	start=max(1,1-offset) end=min(N,N-offset) do i=start,end A(i)=s*A**(i+offset)** enddo

偏移量作为从小到大遍历循环的索引，是负数还是正数有着很大的区别。为正数时，我们认为有一个伪相关，因为流水线计算 A(i) 时需要的 A(i+1) 总是可以获得的，则没有阻塞。然而，真相关时，流水线在计算 A(i) 时必须阻塞直到 A(i−1) 得到结果。这可能导致如图 1-7 中左图所示的一个很大的性能下降。该图显示了不同循环长度下浮点数运算的性能。由于片上高速缓存（cache）延时小和带宽高的特点，这种下降只在高速缓存中很明显。如果循环长度足够大使所有数据不得不从内存中取得，那么流水线阻塞的影响就微不足道了，因为这些额外的时钟周期很容易被内核等待片外数据的时钟隐藏。

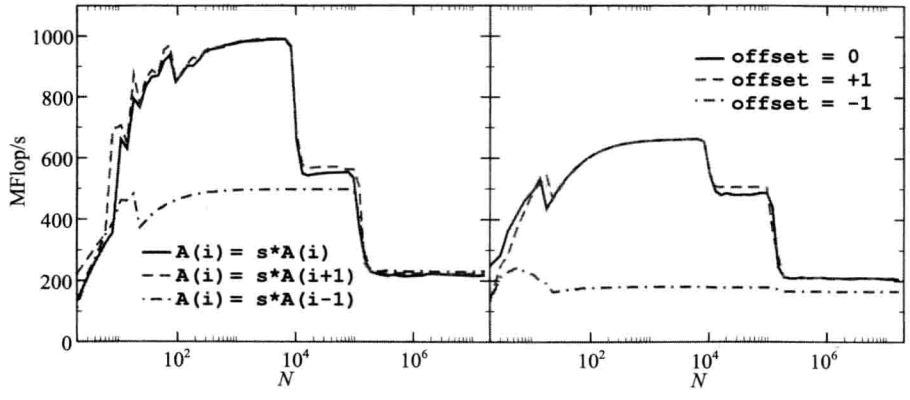


图 1-7 循环偏移量为常量和变量时，分别对循环性能的影响（AMD Opteron 2.0 GHz）

尽管可能有人认为，在编译时是否知道偏移量无关紧要，但图 1-7 中的右图显示出了一个可变的偏移量导致的性能戏剧性下降。在这种情况下，编译器不能使用最佳的软流水或循环优化。这实际上是一种常见现象，而不仅是跟软流水相关。为编译器隐藏信息对性能有着重大影响（这个特殊例子是由于编译器抑制了 SIMD 向量化，见 1.2.4 节和习题 1.2 及 2.2）。

软流水的一些问题与高速缓存的使用有关，详见 1.3.3 节。

1.2.4 超标量

如果处理器被设计成每个时钟周期执行多条指令，或者更一般地说，每个时钟周期可以得到多个结果。在超标量设计的多个细节中可以反映这一目标：

- 可以同时读取和分析多条指令（目前为 3 ～ 6 条指令）。
- 能在多个（2 ～ 6 个）整型运算单元执行地址和其他整数运算，这与上一点有很大的关联，因为需要同时执行代码才能向这些单元提供数据。
- 并行运行多个浮点数流水线。通常有一个或两个加 – 乘运算相结合的管道执行像 a = b+c*d 这样的操作。

13

- 高速缓存足够快能够达到每个时钟周期多个存取操作，可用的加载执行单元的数量（2 ～ 4 个）反映了这一点。

超标量是并行执行的一种特殊形式，一种变形的指令级并行（Instruction Leud Parallelism，ILP）。为了充分利用超流水线，乱序执行和编译优化必须协同工作。然而即使是最先进的体系结构，通过编译生成的代码要达到每周期 2 ～ 3 条指令的吞吐量也是极其困难的。这就是为什么一些性能要求高的应用程序有时使用汇编语言编写。

1.2.5 SIMD

随着 20 世纪 70 年代第一台向量超级计算机问世，SIMD 的概念变得家喻户晓（见 1.6 节），并且成为 20 世纪 80 年代和 90 年代早期大型并行互连机的基本设计原则。

最近许多基于高速缓存的处理器拥有整型和浮点型 SIMD 操作的扩展指令集 [C107]，这使人联想到它们的历史根源，那时只能执行小规模运算。它们利用一个能够装下两个双精度或 4 个单精度浮点数的宽寄存器来实现算法的并行操作。图 1-8 例子中有两个 128 位寄存器，每个都能存放 4 个单精度浮点型数据，单条指令即可一次启动 4 次加法运算。注意，

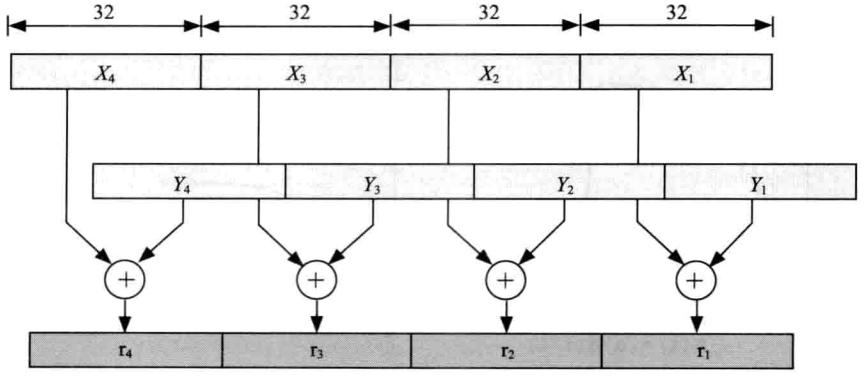


图 1-8 SIMD 例子：有两个 SIMD 寄存器（x, y），每个寄存器的长度为 128 位，用于单精度浮点数的加法。一条指令同时启动 4 个浮点数操作

SIMD 不会关注这些操作的相关性，如果有足够的算术单元可用，这 4 个加法运算就能够真正并行执行，否则被送到单个流水线中。当被送到单个流水线时，SIMD 会减少超标量（和复杂性）但是不会减少峰值运算性能，而当有足够的算术单元可用时，SIMD 操作能较大提升峰值性能。在两种情况下，存储于系统（或至少是高速缓存）必须能够有足够大的带宽使得所有单元能够一直处于工作状态。SIMD 指令集的编程和优化详见 2.3.3 节。 14

1.3 存储层次

数据以不同的方式存储在计算机系统中。前面章节提到，CPU 有一组可以不延时访问的寄存器。另外，有一个或几个容量小但存取速度快的 cache，用以保存最近被访问过的数据项副本。主存要慢得多，但是容量比 cache 大很多。最后，数据能够存放在磁盘上，在需要的时候再复制到主存中。这是一个复杂的层次结构，为了搞清楚它的性能瓶颈，理解数据在不同层次之间的传递原则是至关重要的。下面我们将集中描述从 CPU 到主存的各个层次（见图 1-3）。

1.3.1 高速缓存

高速缓存是低容量、高速度的存储器，通常集成在 CPU 内部。只要认识到主存的数据传输速度远低于 CPU 的运算速度，就能很容易理解为什么高速缓存是必不可少的。当每核的峰值性能达到 GFlop/s 时，存储器带宽即数据从存储器到 CPU 的传输速率，还停留在 GB/s，完全不能使运算单元保持繁忙（更加详细的分析见第 3 章）。更加糟糕的是，为了将一个数据项（一个或两个 DP 字）从主存传送到 CPU，需要一个等待时间，称为延迟。延迟通常定义为传输一个 0 字节消息需要的时间。主存的延迟通常是几百个 CPU 周期，由存储器芯片、芯片组和处理器等不同成分组成。尽管摩尔定律保证了芯片复杂性和性能的稳定增长，但是存储器性能的增长速度仍然在一个很低的水平上。CPU 和主存之间的延迟和带宽有越来越大的差距 [R34，R37]。

在很多情况下，高速缓存能够缓解主存带来的影响。通常至少有两级 cache（见图 1-3）分别称为 L1 和 L2。L1 通常分成两个部分：指令 cache（I-cache，L1I）和数据 cache（L1D）。而在 L2 上，存储数据和存储指令通常都是相同的。一般来说，cache 离 CPU 寄存器越近，它的带宽越大，延迟越低，管理开销就越小。当 CPU 发出读请求（加载）时，传输一个数据项到寄存器中，L1 cache 逻辑检查该数据项是否已经存放在其中。如果在，则称为 cache 命中，且请求能够被立即响应。然而在 cache 不命中的情况下，数据需要从外层 cache 中取出，最糟的情况下需要从主存中获得。如果所有的 cache 都被占用，则需要由硬件实现的算法来进行数据替换，用新的数据替换 cache 中原来的数据。而在 cache 不命中对于写操作更 15
加复杂，后文将介绍。因为代码中往往有很多循环，所以相比于数据 cache，指令 cache 的重要性要稍低，指令 cache 的不命中率与数据高速缓存相比也要低很多。

cache 只有在应用程序的数据使用具有局部性引用时才会对性能产生积极影响。更加具体地说，数据项被加载到 cache 中后，在没来得及被换出前被再次使用，这也称为时间局部性。现在我们运用一个简单的模型来估计高速缓存带来的性能增益。使用一个参数 τ 表示速度上 cache 比主存快的倍数（这涉及带宽和延时；存在更精确的模型，但是计算效果是一样的）。令 β 等于 cache 的重用率，即最近被访问过的数据被再次访问的次数。主存访问时间

（同样这里包括延时和带宽）记为 T_m。在 cache 中，数据访问时间减少至 $T_c = T_m/\tau$。对于一些确定的 β，平均访问时间达到 $T_{av} = \beta T_c + (1-\beta)T_m$，于是我们计算性能增益为：

$$G(\tau,\beta) = \frac{T_m}{T_{av}} = \frac{\tau T_c}{\beta T_c + (1-\beta)\tau T_c} = \frac{\tau}{\beta + \tau(1-\beta)} \tag{1-4}$$

如图 1-9 所示，仅当 cache 的重用率接近 1 时，它才能获得一个显著的性能优势。

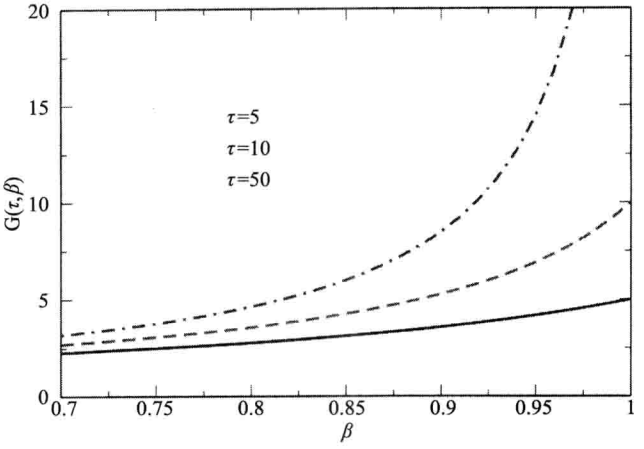


图 1-9 不同的 cache 重用率（β）带来的性能增益，τ 表示 cache 比主存快的倍数

不幸的是，支持时间局部性是不够的。许多应用程序为流模式：大量数据被加载到 CPU、修改然后写回，而没有被及时复用。对于一个仅支持时间局部性的 cache，复用率 β 等于 0。由于 cache 中的旧数据项被新的取代，产生很大延迟，因此每一次新的数据加载都有很大开销。为了减少延时开销，在 cache 中加入了一个叫 cache 行（cache line）的特殊组织结构。在 cache 和主存之间的数据传输，都在 cache 行级别上进行（这个规则可能有一些例外，可参考下面介绍的非时效性存储的内容）。cache 行的优势在于，cache 不命中的延时开销只发生在数据项第一次不命中的时候。cache 行作为整体从主存中取出，相邻数据项能够以一个更低的延时从 cache 中取出，提升了 cache 的命中率 γ，不要与重用率 β 混淆。因此，如果应用程序具有一些空间局部性，即连续访问相邻数据项的概率越高，那么延时能够显著减少。cache 行的缺点是不支持不稳定的数据访问模式。那样会导致每次数据加载都会产生不命中和随之而来的延时，而且由于传输了整个 cache 行会导致主存总线上传输很多从未被使用过的数据，则应用程序的有效带宽将非常低。然而从总体上来看 cache 行的优势还是很受欢迎的，大部分处理器制造商都使用这一机制。

16

假设一个基于 DP 浮点型的流式应用程序在 CPU 上执行，cache 行的长度为 $L_c = 16$ 字，空间局部性的命中率 γ 为一个看似很大的值上：$\gamma = (16-1)/16 = 0.94$。它的性能仍然依赖于主存的带宽和延迟，即代码受制于内存。为了使应用程序能够真正受制于 cache，而不再受主存带宽和延时的影响，γ 必须足够大，从而使得处理 cache 中数据的时间远远大于重新重加载数据的时间，这种情况的发生取决于执行操作的细节。

现在，我们可以定性分析基于 cache 体系结构上三维向量算法性能，如图 1-4 中所示。在很小的循环长度下，处理器流水线太长以至于效率不高。随着 N 的增加，这种制约变得微不足道，而当 4 个数组都能在最内层 cache 命中时，性能达到一个饱和值，这个值由一级

（L1）cache 带宽和处理器发出存取指令的能力决定。继续增加 N 的大小将导致性能急剧下降，因为最内层 cache 的容量不足以容纳所有数据。二级（L2）cache 总有更长延迟但带宽只是接近于一级 cache，因此延迟比预期更大。然而，来自 L2 的数据流还带来一个缺点：此时的 L1 不得不向寄存器提供数据，同时又持续地重载并数据写回到 L2，这限制了 L1cache 的带宽利用率。由于 cache 向其他存储层次传送数据的能力高度依赖体系结构，除了最内层 cache 和主存外，整体性能很难被估计。随着 N 的增加，不同层次 cache 的性能分别下降，直到最后，甚至最外层的 cache 容量都显得过小，以致所有的数据不得不与主存交换。而带宽瓶颈的位置也与 cache 的容量直接相关。一些基本参数，例如 cache、主存带宽和应用程序数据需求等，对循环的性能有着不同影响，3.1 节将具体讲述怎样预测这些参数带来的性能影响。

写数据比读数据更加复杂。在当前的 cache 中，如果即将被写回的数据已经存储在 cache 中，则发生写命中。写 cache 有多种策略，但是通常最外层的 cache 采用写回策略：先修改 cache 行，当数据要被从 cache 中换出时，整个数据再被写回主存。然而，当写不 17
命中时，在数据被修改之前，由于 cache 和主存的一致性，需要先将数据从主存写入 cache 中，这就是所谓的写分配，这将导致从 CPU 到主存的写数据流会使用总线两次：一次是从主存导入数据到 cache 行，另一次是修改后写回（由于写分配，traid 基准测试代码的数据转移需求增加了 25%）。因此，流式应用程序不总是受益于写回策略，其可能导致更多写分配的发生。所以要尽可能避免写分配的发生，某些体系结构提供了这种功能，并且一般有两种不同的策略：

- 非时效性存储。有一些特殊的存储指令不必通过所有的 cache 层次而将数据直接写进内存中。写数据流不会“污染”cache，但是这会导致缺乏时间局部性。为了避免大量的延时，通常需要有一个小的写缓冲区，临时存储着由许多非时效性存储的数据[C 104]。
- cache 行标零。一些特殊的指令能把 cache 行标零，代表发生过改变。这些数据在替换出去的时候要写到内存。与非时效性存储比较，cache 行标零方法能够为写数据流用完 cache 空间。另一方面，在高速缓存受限的条件下，存储操作不必减速。但是必须注意，即使只有一部分数据改变，整个 cache 行在替换的时候都要写到内存。

这两种方法都能被编译器使用，程序员也可以通过指令来指示编译器如何做，在简单情况下，编译器在代码优化阶段会自动使用这些指令。但是要留意，使用非时效性存储会让受限于 cache 的代码变慢，但是会让受限于存储的代码则变得更有效。

需要注意的是，之所以需要写分配，是因为 cache 和内存交换的基本单元是 cache 行。还有一种普遍的误解认为写分配只需要保持多处理器核的 cache 一致就可以了（还需要与内存一致）。

1.3.2 高速缓存映射

之前我们默认假设了 cache 行与存储单元之间的映射没有任何约束，这种设计也叫全相联（fully associative）。不幸的是，这种方式设计的 cache 很难做得很快或者很大，因为全相联的 cache 有很大的簿记开销：每一个 cache 行，在 CPU 的地址空间中必须存储其地址，而且每一次存取操作都要检查所有的地址。此外，如果 cache 满了，cache 的替换策略由硬件

实现。最近最小使用策略（Least Recently Used, LRU）会保证“最久”的项被替换，而像最
18 近不使用算法（Not Recently Used，NRU）或者随机替换策略则不能够保证。

直接映射策略则是最直接、最简单的方法。直接将大小为 cache 容量的内存块映射到 cache（如图 1-10），相距 cache 大小整数倍的存储单元会被映射到 cache 中相同的行，只要去除最高有效位，就能很快地得到某内存地址的 cache 行。而且 cache 的替换不需要任何算法，不需要硬件支持也没有时间周期的开销。

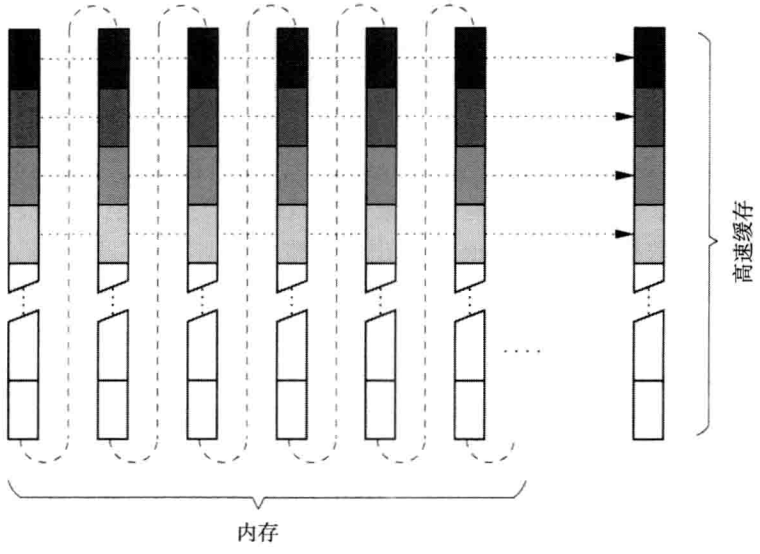


图 1-10 直接映射 cache。相距 cache 大小整数倍的存储单元会被映射到相同的 cache 行（图中阴影方格）

直接映射 cache 的一个缺点是很可能产生颠簸，即某些 cache 行被快速地导入和替换。当应用程序中使用的很多数据映射在 cache 的同一行时就会发生种现象。用一个简单的例子说明，例如还是双精度的跨步三维向量算法，只要改变一下内循环，如下所示：

```
1 do i=1,N,CACHE_SIZE_IN_BYTES/8
2   A(i) = B(i) + C(i) * D(i)
3 enddo
```

用 cache 的容量（双精度字为单位）作为跨步，连续的循环迭代会取相同 cache 行数据，即使每一次导入都填满 cache 行，也会导致一次 cache 失效。原则上 cache 中有大量的空间未被利用，这种情况叫做冲突缺失。如果跨步为 cache 行的长度，即使 N 很小，cache 的利用率也为 100%。而跨步为 cache 容量时，无论 N 多小，cache 重利用率几乎为 0。

19 为了降低替换开销同时减少冲突失效和 cache 颠簸，可以使用组相联。将 cache 分为 m 组相同大小的直接相连 cache，即叫做 m 路组相联。m 的数量是一个存储单元可以被映射到的不同 cache 行的个数（图 1-11 是一个 2 路组相联的例子）。在每一次存取中，硬件很少干涉数据应该存储到哪一路 cache 或者在未命中的情况下哪一路的 cache 行应该替换。

为了在低延迟和防止颠簸中寻求平衡，系统程序员必须考虑 cache 层次。最内层的 cache(L1) 相比于外层 cache 较少使用组相联。典型组相联的组数在 2 ～ 48 之间。但 cache 的有效容量还比较小，例如在应用程序中，根据数据流的数目和它们的跨度、相互位移会产生时间、空间局部性，但能有效利用这些局部性的 cache 部分非常小。

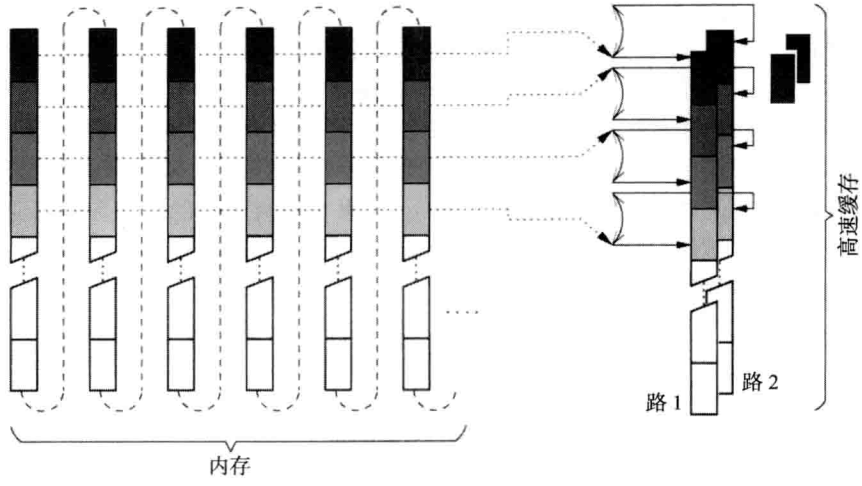


图 1-11 *m* 路组相联 cache，相距为 1/*m* 倍 cache 容量的内存单元将被映射到任意一路 cache 行中（这里 *m*=2）

1.3.3 预取

即使引进了 cache 行以利用空间局部性，可以使 cache 更有效，但是在第一次不命中时还是会有大量延迟。图 1-12 是求向量范数内核的例子。

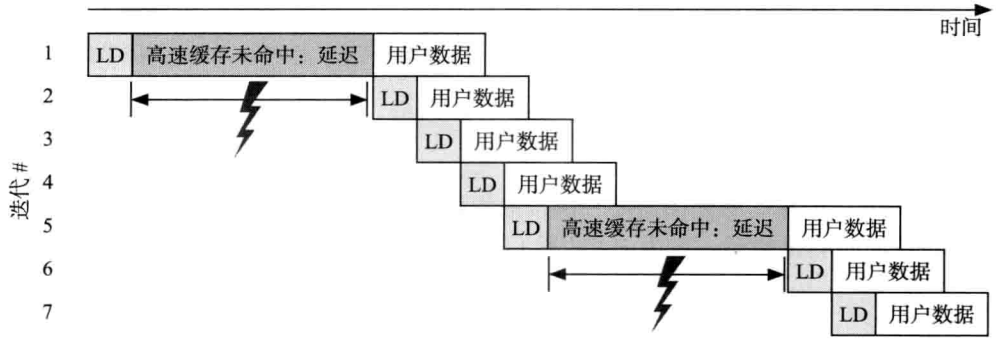


图 1-12 求向量范数循环时，cache 未命中时的时间表以及延迟。每一次未命中都会有延迟

```
1 do i=1,N
2   S = S + A(i)*A(i)
3 enddo
```

代码中只有一条数据加载流。假设 cache 行的长度为 4，则在三次加载中能在 cache 中命中，然后就会发生一次 cache 不命中，这种长延迟会导致内存总线长时间空闲。 20

增大 cache 行的长度会减少上述类似的延迟，但是会造成访问模式不稳定而减慢应用程序的执行速度。为了平衡，主流的 cache 行长度为 64～128 字节（8～16 个双精度浮点数）。到目前为止还会存在较大的类似延迟，所以内存带宽、内存总线利用率较低。假设一个商用系统存储延迟为 50ns，带宽为 10GB/s，传输一个 128 字节的 cache 行需要 13ns，则 80% 的总线带宽没有被使用，此时延迟已经是一个比带宽更重要的问题。

有很多方法可以减少延迟，预取是其中之一。预取是指在应用程序使用数据之前就已经

把它们导入 cache。通过软流水，编译器会在程序中打乱一些指令，从而让硬件有时间异步地把数据提前导入 cache（如图 1-13）。这里假设现代系统架构一定程度上能支持异步传输。还有一种方案可以达到预取一样的效果，一些处理器有一个硬件预取器，能够根据数据访问模式提前读取应用程序数据，保持持续的数据流，并提供与预取指令相同的服务。无论处于哪个阶段，必须强调的是预取使用的资源必须由设计进行限制。存储子系统必须能支持一定数量的预取指令（例如正在响应的预取请求），容忍存储流水线的阻塞和不可避免的延迟。我们可以估计为了隐藏所有延迟而预取需要的指令数量，假设 T_ℓ 是延迟，B 是带宽，则传输 L_c（以字节为单位）需要的时间：

$$T = T_\ell + \frac{L_c}{B} \tag{1-5}$$

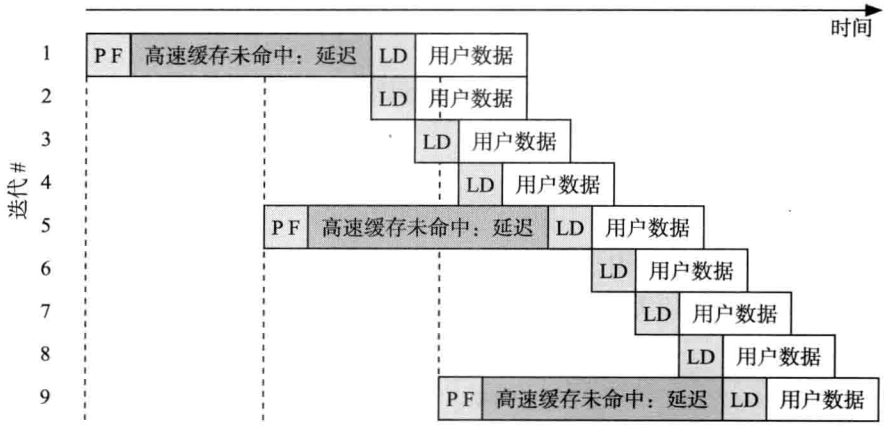


图 1-13 通过预取，计算和数据传输过程能够更好地重叠。此例中为了完全隐藏延迟需要预取两条指令

21 每次 cache 行传输的时候都需要一次预取操作，处理器需要预取的指令数量是在时间 T 内能传输的 cache 行的数量，设为 P（见图 1-13），则处理器必须：

$$P = \frac{T}{L_c/B} + = 1 + \frac{T_\ell}{L_c/B} \tag{1-6}$$

例如 cache 行长度为 128 字节 (16 个双精度浮点数)，B = 10GB/s，T_ℓ=50ns，则可以得到 $P \approx 5$。如果没达到这种预取要求，延迟不会完全隐藏，存储带宽也不能完全被利用。另一方面，如果应用程序使用很多高速缓存块数据（这些数据在传输时不能被隐藏）的浮点型操作，执行时间比数据传输的时间明显要长，将不会受到带宽的限制，对存储子系统的压力也不大（3.1 节中会介绍合适的高性能模型）。这种情况下，不需要太多指令预取。

严重依赖存储带宽的应用程序可能会给预取机制带来较大压力。可以用一个共享存储总线的协处理器来提供预取功能，减轻带宽的压力（详细请参考 1.4 节的多核设计）。通常，如果程序具有流模式访存，那么一个好的编程模型应该提供一种更长的持续数据流方法。

最后，有些要注意的地方。图 1-12 和图 1-13 强调了指令预取对隐藏延迟的作用，但是带宽的性能限制也不能忽视。即使单个 cache 行的传输时间主要是传输延迟，预取也不能提
22 高主存带宽。

1.4 多核处理器

近年来并且至少在接下来的十年，虽然摩尔定律依然成立，但标准的微处理器开始遇到散热问题：数百万个晶体管芯片的开关和漏电功耗如此之大，使散热成为一个工程和商业主要关注的问题。另一方面，由于硬件架构方面的改进和 cache 尺寸的增加，促进了时钟频率不断提高，但已不足以获得符合摩尔定律 1 比 1 的性能提升。处理器厂商们正在寻找一种突破这种能耗性能瓶颈的新方法——多核设计。

对于半导体处理技术，CPU 的功耗和时钟频率 f_c 的三次方成比例（实际上是 f_c 和电源电压 V_{cc} 的二次方的乘积，但是 f_c 与 V_{cc} 成比例），所以降低频率和电压就可以明显地降低功耗，这是多核技术背后的动机。假设有一个单核时钟频率是 f_c，性能是 p，功耗是 W，时钟频率的微小变化（$\varepsilon_f = \Delta f_c/f_c$）会引起性能方面的变化（$\varepsilon_p = \Delta p/p$），在所有其他条件一样的条件下，$|\varepsilon_f|$ 是 $|\varepsilon_p|$ 的上限，这也要考虑具体的应用。功率消耗为：

$$W + \Delta W = (1 + \varepsilon_f)^3 W \tag{1-7}$$

从式（1-7）中可看出，在保证功耗不变的前提下，降低时钟频率创造了将多个 CPU 核放在同一个晶片（更一般地为同一个组件）上的可能性。而对于 m 个“慢”核，这个条件表示为：

$$(1 + \varepsilon_f)^3 m = 1 \Rightarrow \varepsilon_f = m^{-1/3} - 1 \tag{1-8}$$

23

这些核和速度更快的核有相同的晶体管数量，但是根据摩尔定律，增加晶管的数量并不增加成本。图 1-14 显示了频率与处理器核的数量关系。所有多核芯片的总性能表示为：

$$p_m = (1 + \varepsilon_p) pm \tag{1-9}$$

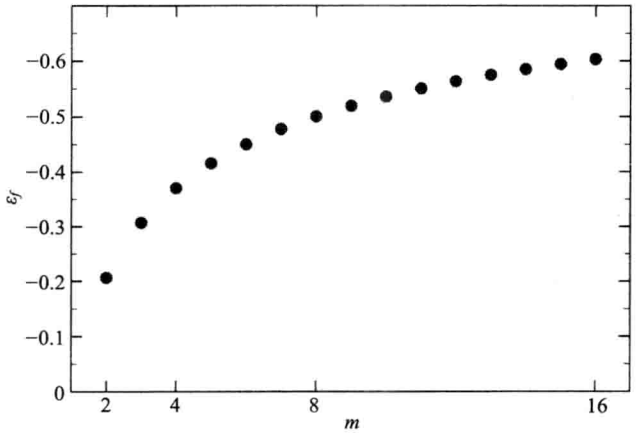


图 1-14 在给定的工艺技术和功率上，所需的相对频率与多核芯片上核的数量的关系

而多核芯片的总性能必须大于单核的性能：

$$\varepsilon_p > \frac{1}{m} - 1 \tag{1-10}$$

当然，由于现在的制造技术，增加 CPU 晶片是不难的。因此，实现多核最简单的方法就是将各个 CPU 晶片放在一个共同的组件内。比如更小的结构长度等制造工艺方面的改进，将能够在一个晶片上安置更多的核。但是相比前一代产品，多核芯片上的单核性能可能会有所下降，否则芯片上晶体管的数量和时钟频率都会下降。有些公司甚至不惜以可能会带来新

的编程模型为代价，而采用更为激进的方法，即设计更简单的核。

1.9节中过于乐观的假设是 m 个核会表现 m 个单核性能总和的假设仅在极少数情况下才成立。然而到目前为止，多核结构已经被所有主要的处理器制造商采用。我们将同时使用核，CPU和处理器的概念，避免产生误解。插槽（socket）指的是封装了多个核（有时可能是多个芯片）的物理单元，通常配备引脚来使其成为可更换的部件。传统的台式电脑，只有一个插槽，但是标准的服务器有两个或者四个，它们共享相同内存。4.2节将详细介绍共享
24 存储的并行计算机体系结构。

片上和插槽上的核组织结构有显著的差别：

- 一个晶片上的核有各自的cache或者共享特定层次的cache（图1-16～图1-18）。在后面提到时，我们称共享了特定cache层次的一组核为cache组（cache group）。例如，图1-17中的六核CPU分成了6个L1组，3个双核L2组和一个包含整个插槽的L3组。

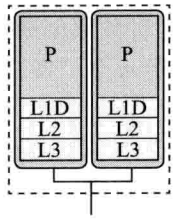


图1-15 具有独立L1、L2、L3三层的cache的双核处理器芯片（Intel“Montecito”）。每个核在各cache层上都有自己的cache行

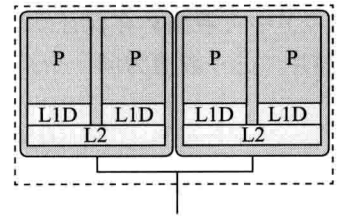


图1-16 两个双核处理器组成的四核处理器芯片（Intel“Harpertown”）。每个双核处理器具有共享的L2 cache和独立的L1 cache，共有两组双核L2高速缓存

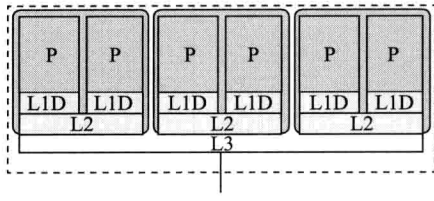


图1-17 六核处理器芯片。每组核具有单独的L1、共享的L2和一个整个核共享的L3 cache。L2 cache组为双核处理器拥有，L3 cache组为整个处理器拥有

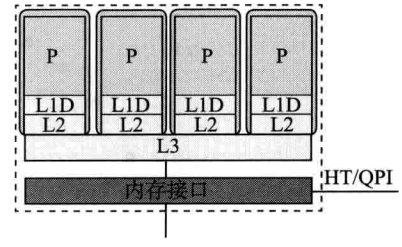


图1-18 四核处理器芯片。具有单独的L1和L2 cache，一个共享的L3 cache（AMD“Shanghai”和Intel“Nehalem”）。有四个单核L2 cache组，一个全局的L3 cache组。有一个内建的内存接口，不需要芯片就可以直接连接内存和其他插槽

共享cache使得核与核间的通信不通过主存，可以非常明显地降低延迟并以指数级增加带宽；而其负面影响可能是带宽瓶颈。共享或独立cache对性能的影响高度依赖于代码和系统。接下来的章节会更多地讨论这个问题。

- 最新的多核设计集成了一个内存控制器，内存控制器可以直接连接内存模块，而不

使用独立的逻辑电路（芯片）。这减少了内存延迟，也使像超传输（HyperTransport，HT）和快速通道（QuickPath，QPI）等互联网络快速增加。

- 在 cache 之间可以存在快速数据通路，使得高效 cache 一致性通信成为可能。

多核的出现得到的一个最重要结论就是在并行编程时需要尽可能大地利用这些并行资源，而不是只依靠单核的性能。因为单核性能提高的空间不大了，仅根据摩尔定律改进 CPU 来提高处理速度可能不太现实。第 5 章将描述并行编程的限制和规则。在 4.2 节中将呈现更多关于双核和多核的设计细节，如共享内存体系结构等。第 6 章和第 9 章给出了当前工程和科研领域使用的主流并行编程范例。 25

多核结构带来的另一个挑战是每个单核可用的主存带宽和 cache 容量在减少。虽然生产厂商通过使用更大容量的 cache 来弥补，但是一些算法性能总是受到主存带宽限制，由于存在总线竞争，使共享内存总线的多个核性能下降。在编程中，减少访存次数和有效使用带宽成为了使用摩尔定律进行有效编码主要考虑的问题。第 3 章介绍了在这种情况下一些有效的技术。

最后，在多核芯片上，共享存储与非共享存储有着不同的复杂结构（见图 1-17、图 1-18），使得不同核之间的通信也非常不同：如果存在一个共享的 cache，它们可以通过 cache 中的一个变量来进行同步，而不必通过内存总线来交换信息（参见 7.2 节与 10.5 节中的实际交换），这样两个核可以更快地交换信息。在写这本书时，几乎还没有多核的编程技术能够利用这种特征来提高并行代码性能。

因此，根据运行应用程序的通信特征和带宽要求，多线程或进程在多核中（也可能是多插槽）的运行环境非常重要。对于怎样在硬件和程序（线程、进程）中建立紧密联系，附录 A 中会详细描述。本书经常提到性能与并行程序之间紧密关系的重要性，例如 6.2 节、第 7 章、第 8 章和 10.5 节。

1.5 多线程处理器

所有现代的处理器都以高度流水线化来提高性能（如果可以使用流水线）。前面提到，一些因素会影响流水线的高效利用：相关性、存储延迟、不确定的循环长度、指令混合以及分支判断错误等（参考 2.32 节）将导致流水线频繁等待，很大一部分执行资源处于空闲状态（见图 1-19）。不幸的是，这种情况是规则而不是意外。为了提高时钟频率而尽可能设计长流水线会增加算法的复杂性，结果导致没有获得成比例的性能提升，处理器也会有更多的功耗。

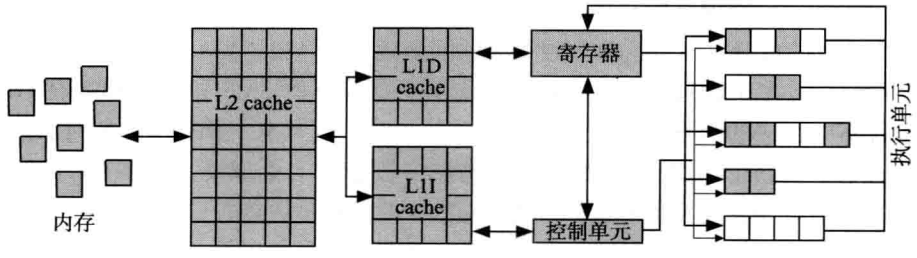


图 1-19 没有使用 SMT 的（多）流水线微处理器上的数据流和控制流简图。执行单元中的白色框图代表流水线气泡（阻塞周期）。图形由 Intel 提供

正是由于这个原因，到很多现代的处理器设计中加入了线程，也叫做超线程技术（Hyper Threading）[V108，V109] 或者实时多线程（Simultaneous Multithreading，SMT）。
26 这种设计的特点是 CPU 核的结构状态多次复现在不同的线程中，结构状态包括数据、状态和控制寄存器，还有栈和指令指针，但诸如算术运算器、cache、内存接口等执行资源没有重复。由于多个结构状态的存在，CPU 看起来像是包含了一组核（有时也叫逻辑处理器），可以并行执行多个指令流或者线程，而不用理会它们是否属于同一个程序。硬件必须记录指令属于哪个结构状态。所有线程共享这些执行资源，所以由流水线阻塞而产生的气泡可以用另外一个线程的指令来填充。如果存在并行运行的多个流水线（参见 1.2.4 节），一个线程搁置了或者多个流水线正处于空闲状态，则另一个线程就可以使用它们，参见图 1-20。

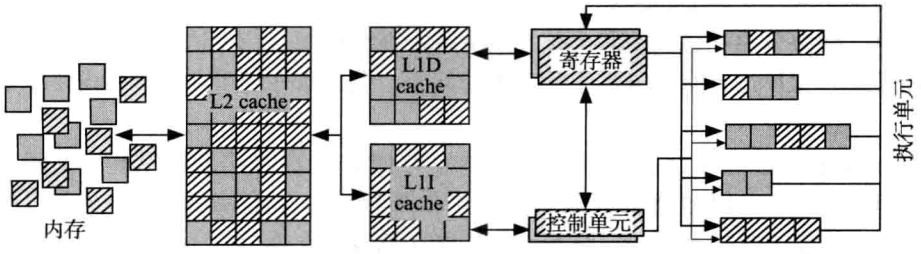


图 1-20 （多）流水线微处理器上的数据流和控制流简化的框图，具有二路 SMT。两个指令流（线程）共享 cache、流水线等资源，但是具有其各自的结构状态（寄存器、控制单元等）。图形由 Intel 提供

SMT 可以通过多种不同的方式实现。这些方式的一个不同点是在流水线上线程如何实现切换。理想的情况切换按周期发生，但是很多实现方式需要将流水线全部清空来支持另外一个线程，这会带来非常大的延迟。

27 如果多个线程的代码既可以发射在同一流水线内也可以发射在不同的流水线上，则 SMT 可以提高指令的吞吐量。如果不同的线程使用不同的执行资源，比如浮点运算和整数运算，就很容易提高吞吐量。一般以浮点操作为主的科学计算，经过很好的优化后，并不能从 SMT 中获益太多，但是也有例外：在一些架构上，内存引用表的数量与现线程数量一致时，同时运行足够多的线程就能够充分利用主存带宽。

如果对于拥有 SMT 的线程资源有限，单个指令流的性能不会提高，甚至会有小小地下降。并且，多个线程共享很多资源尤其是 cache 时，如果代码对 cache 容量敏感就很可能增加 cache 容量冲突（由高速缓存容量较小引起的容量冲突）。最后，SMT 会严重增加同步操作的开销：如果同一物理核执行的几个线程都通过执行紧凑的自旋等待循环等待某些事件，它们将竞争共享的执行单元，这将导致很大的同步延迟。[132，133，M41]

如果系统上有多个物理核，并且操作系统和程序员了解 SMT 机制，那么在不同物理核上默认运行不同程序的线程和进程是一个不错的想法，但只有在确保安全时才使用 SMT 来提高总体性能。SMT 的出现，仿射机制将比在多核芯片上更重要（参考 1.4 节和附录 A）。对于手头的应用程序，需要彻底测试才能确定 SMT 机制是否能提高性能。如果不能在一个物理核上通过合适仿射应该只保留一个逻辑核，并且如果可能的话 SMT 应该被一起关掉。

1.6 向量处理器

从 Cray 1 超级计算机开始，直到基于 RISC 的高度并行计算机出现之前，向量机一直占

据着科学计算的主要领域。在写这本书时，只有两家公司还在制造和销售向量机。但因为对内存带宽和运行时间有高度需求，向量机还是有着一个充满商机的市场。

根据设计，对于合适的可向量化的代码，向量处理器相较于标准的微处理器可以达到一个较好的实际性能。这种设计遵循单指令多数据（SIMD）的范例，即一条简单的机器指令被自动地应用于很多类型相同的参数。许多现代的基于 cache 的微处理器以扩展 SISD 指令集的形式采用这些技术（细节参考 2.3.3 节）。而且向量机在执行单元和存储子系统上有大量的并行操作。 28

1.6.1 设计原理

现代向量处理器与 RISC 设计非常像，都是寄存器—寄存器型的机器：机器指令运行在向量寄存器上，每个向量寄存器存储长度在 64 ～ 256（双精度）之间的一些参数。向量寄存器的宽度称为向量长度 L_v。对于每一种算术运算，像加法、乘法、除法等都分别有一条流水线，每一条流水线在每个指令周期都可以得到一定数目的结果。对于乘法和加法流水线来说，得到 2 ～ 16 个结果，这也称作多轨流水线（multitrack pipeline）（参考图 1-21）。其他像平方根和除法操作比较复杂，流水线的输出要低很多。但是即使只有单一流水线的向量处理器也可以达到基于 cache 的超标量微处理器相同的峰值性能。为了向向量寄存器提供数据，有一个或者多个直接跟主存相连的读取、存储、读取与存储相结合的流水线。尽管最近像 NEC SX-9 的设计引进了容量小的片上存储，但传统的向量 CPU 没有缓存层次的概念。

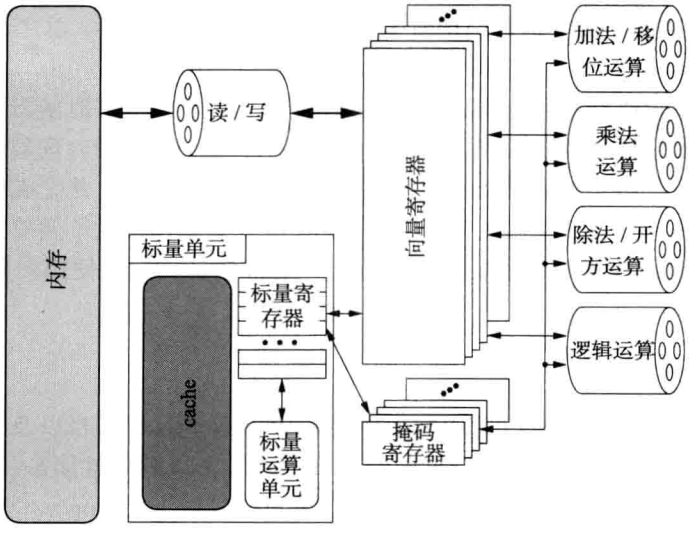


图 1-21 具有 4 轨流水线的典型向量机简图

为了在向量 CPU 上获得合理的性能，必须采用 SIMD 类型的指令。我们来看一个简单的例子，有两个数组 A(1 : N) = B(1 : N) + C(1 : N)。在一个基于 cache 的微处理器上，这个运算最终的实现是在 A、B 和 C 上的一个循环（可能会软流水），对于每一次计算，必须要 29
执行两次读取操作、一个加操作和一个存储操作，并且还必须有整型和分支逻辑来实现循环。如果数组的长度比寄存器长度短，则向量 CPU 可以对于整个数组使用一条指令：

```
vload V1(1:N) = B(1:N)
vload V2(1:N) = C(1:N)
vadd V3(1:N) = V1(1:N) + V2(1:N)
vstore A(1:N) = V3(1:N)
```

在这里，V1、V2、V3 表示向量寄存器。追踪分散在不同的流水线上的向量索引的工作是自动完成的。如果数组的长度比向量的长度大，循环就必须以向量长度为单位分块执行：

```
do S = 1,N,Lv
   E = min(N,S+Lv-1)
   L = E-S+1
   vload V1(1:L) = B(S:E)
   vload V2(1:L) = C(S:E)
   vadd V3(1:L) = V1(1:L) + V2(1:L)
   vstore A(S:E) = V3(1:L)
enddo
```

这个工作由编译器自动完成。

像向量加这样的操作并不需要等到向量寄存器将所有参数都准备就绪才开始运算，而是可以在最初的一些参数就绪之后就可以开始执行。这个特征称为链接（chaining），这也是不同管道（例如乘法、加法）能够同时操作的必要条件。

很明显，在 RISC 出现前向量结构明显地降低了指令发射速率的要求，那个时候多发射超标量处理器还没有足够快的指令 cache。更重要的是，读取 / 存储操作的速率需要和 CPU 核频率匹配，所以为运算流水线提供数据就更不是问题了。由于现代内存芯片在一次缓存操作后需要几个时钟周期的恢复时间（也称为 bank 忙碌时间），所以可以通过有大量的 bank 结构的内存布局来实现两者速率的匹配。为了减小两者之间的差距，现代向量机提供数以千计的内存 bank，这也导致了这种结构对于通用计算来说相当昂贵。总之，向量处理器是通过高度并行的流水线以及高带宽的内存访问来获得它的性能。

编写程序以使编译器产生有效 SIMD 向量指令称为向量化。有时候需要代码重构或者在源代码中插入指令指针来帮助编译器确认 SIMD 并行。每一个向量处理器都有一个单独的标量单元，用来执行那些不能向量化的代码（接下来的章节中讨论）并完成任务管理工作。向量处理器中的标量单元要比标准的 RISC 或基于 x86 设计中的标量单元差很多，所以为了获得高性能，向量化就显得格外重要。如果代码不能被向量化，则使用向量机不会带来任何好处。

30

1.6.2 最高性能估计

向量处理器的峰值性能可以通过加法和乘法流水线的 track 数目以及时钟频率得到。比如，一个 2GHz 的向量处理器以及具有 4 个 track 的流水线，峰值性能是：

$$2(\text{加法}+\text{乘法})\times 4(\text{track})\times 2(\text{GHz})=16\ \text{GFlop/s}$$

求平方根、除法和其他操作由于有着较差的吞吐量，对计算峰值性能没有较大的贡献，所以在这里不予考虑。关于内存带宽，有 4 个 track 的 LD/ST（参见图 1-21）流水线可以得到的读写带宽为：

$$4(\text{track})\times 2(\text{GHz})\times 8(\text{B})=64\ \text{GB/s}$$

这恰好是 NEC SX-8 处理器的标准规格。和基于 cache 的标准微处理器相比，向量处理器的内存接口的频率通常和核相同，能够为峰值性能提供更高的带宽。注意到上面这些计算都是建立在一个假设上：向量处理单元一定会被用到——如果代码是不可向量化的，因为要受标量单元的限制，所以不管峰值性能或者峰值存储带宽都不能达到。

通常对于一个拥有简单内存访问类型的循环，其性能可以预测。第 3 章将会对平衡分析给予详细介绍，例如对结构和循环代码的特点进行性能预测。对于向量处理器，由于不存在 cache，所以预测一般会比较简单。以代码清单 1-1 为例，3 次读取操作，1 次存储操作和两个浮点操作（加法和乘法）。由于只有单一的 LD/ST 管道，读取和存储操作，甚至对不同数组的读取操作都不能够重叠。但它们可以重叠算术管道并链接到算术管道。在图 1-22 中，长平行四边形代表在向量寄存器上的一个操作，标志着管道操作的执行（与图 1-5 中的时间线很像）。首先必须向一个向量寄存器中从数组 C 读取数据，LD/ST 管道开始于用数组 D 中的数据填充向量寄存器，乘法管道就可以开始在 C 和 D 上执行算数运算。只要来自 B 的数据可用，加法管道就可以计算出最终结果，继而 LD/ST 管道将结果存储到内存。

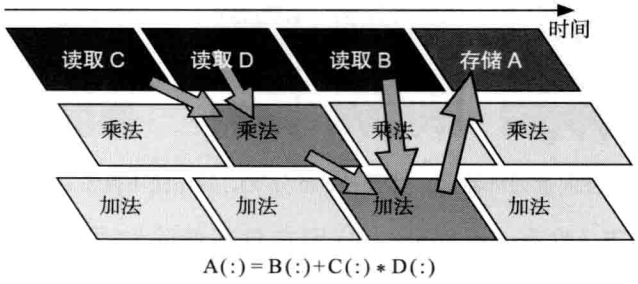


图 1-22　在图 1-21 展示的向量处理器上执行三维向量算法（见代码清单 1-1），使用了流水线后的时间线。浅灰色四边形代表没有使用的运算单元

整个过程的性能瓶颈很显然是 LD/ST 流水线。如果给予合适的代码，硬件能够在相同时间内执行 4 倍的乘法和加法指令（图 1-22 中浅灰色菱形），所以代码清单 1-1 的性能只能达到峰值性能的 25%，在上面描述的向量处理器上性能为 4 GFlop/s，这与图 1-4 中的 SX-8 N 很大时的曲线完全吻合。需要注意的是，由于向量系统上有很大的内存延迟，所以这个限制只对相对大的 N 值可以达到。另外，除了不可向量化的代码，短循环是第二大影响这些结构的性能因素。

1.6.3　程序设计

向量化的必要条件是在循环的迭代之间不存在真数据相关。软流水（见 1.2.3 节）也同样不能出现真数据相关，例如允许向前引用（forward reference）但是向后引用（backward reference）会影响向量化。更精确地讲，真相关的位移间隔必须大于某一阈值（至少是向量的长度，有时更大），这样前面向量操作的结果才会是可用的。 31

因为与“单条指令”的编程规范冲突，内部循环中的分支也会影响向量化。但是我们有一些方法来支持向量化循环中的分支：

- 掩位寄存器（mask register，本质上是向量长度的布尔寄存器）用来实现循环迭代的选择性执行。我们来看下面一个例子：

```
1 do i = 1,N
2   if(y(i) .le. 0.d0) then
3     x(i) = s * y(i)
4   else
5     x(i) = y(i) * y(i)
6   endif
7 enddo
```

32 首先，使用逻辑流水线根据分支条件产生一个每位为布尔类型的向量。接下来这个向量被用来从 if 或者 else 分支中选择结果（见图 1-23）。当然，如果有开销大的操作，那么所有的循环分支都被执行显然是一种浪费，但是向量化的利大于弊。

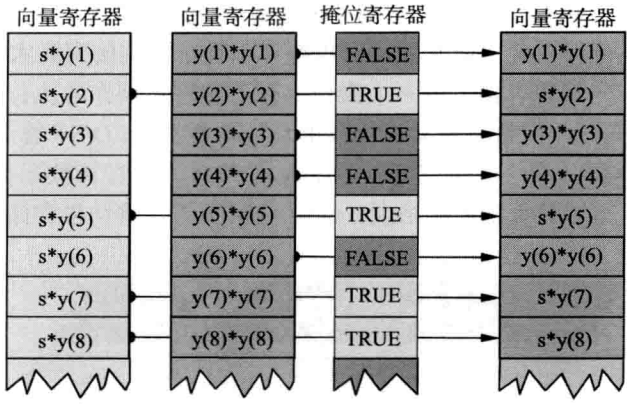


图 1-23 在向量处理器上，具有 if/else 分支的循环使用掩位寄存器向量化

- 对于单个分支（没有 else 部分），特别在包括像除和平方根这些操作的情况下，gather/scatter 是一种向量化的有效方法。在下面的例子中，如果分支预测大部分情况为假时，像图 1-23 中那样使用掩位寄存器会浪费很多计算资源：

```
1 do i = 1,N
2   if(y(i) .ge. 0.d0) then
3     x(i) = sqrt(y(i))
4   endif
5 enddo
```

与掩位寄存器不同（参见图 1-24），所有必要的元素首先被收集到向量寄存器中（gather），然后执行向量操作，最后执行的结果被存储回去（scatter）。

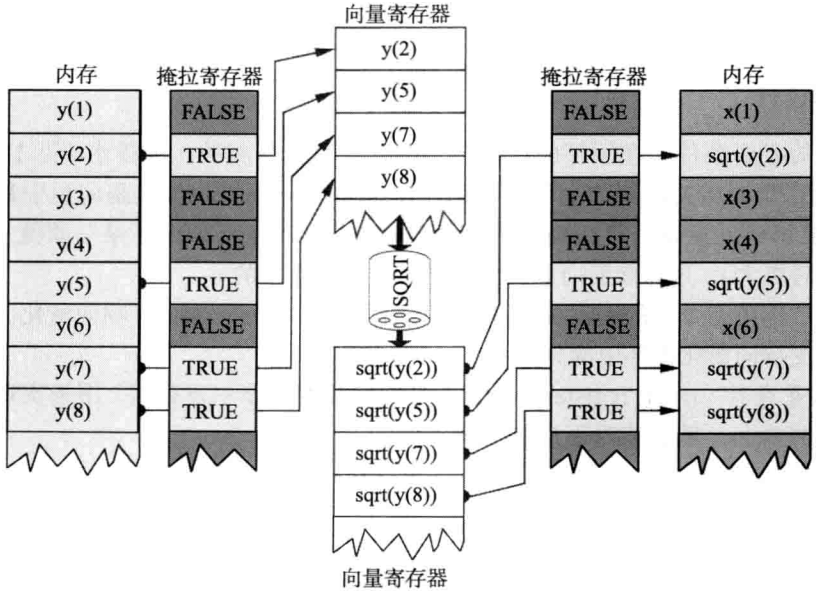


图 1-24 gather/scatter 方法的向量化。只有当掩位寄存器对应位置的元素为真时才能通过主存进行数据传输，数据的读取和存储用的也是相同的一套掩码

编译器会自动执行向量化操作（可能是源代码直接支持向量化）或者代码被重写使得可以显式地使用临时数组来保存所需的向量数据。还有另一种替代方案，使用列表向量，它是一个整型的向量数组，保存着条件为真的索引，通过间接访问，这些索引可以用来重构原始循环。 33

由于 cache 行的概念，基于 cache 处理器对 gather/scatter 操作开销非常大，而向量处理器能更经济地执行 (即使跨度为 1 的访存模式更有效)。

关于向量结构的编程和优化可以从生产商处获得更多参考文档 [V110，V111]。

习题

1.1 *除法的速度*。写一段代码对下列函数积分：

$$f(x)=\frac{4}{1+x^2}$$

x 从 0 ～ 1，结果应该是 π 的近似值。用一个简单的矩形积分就可以实现，即矩形宽为 x_i，步长 Δx，高为 $f(x_i)$，对面积累加：

```
1 double precision :: x, delta_x, sum
2 integer, parameter :: SLICES=100000000
3 sum = 0.d0 ; delta_x = 1.d0/SLICES
4 do i=0,SLICES-1
5    x = (i+0.5)*delta_x
6    sum = sum + 4.d0 / (1.d0 + x * x)
7 enddo
8 pi = sum * delta_x
```

完成程序段，选择合适的 Δx，判断结果是不是 π 的近似值，并计算性能，结果单位为 MFlop/s。假设浮点除法不能被流水线运行，试估计延迟为多少时间周期。

1.2 *数据依赖*。在 1.2.3 节我们讨论了流水线，请看以下代码：

```
1 do i = ofs+1,N
2   A(i) = s*A(i-ofs)
3 enddo
```

s 是一个非零的双精度浮点标量，ofs 是一个正整数，A 是一个长度为 N 的双精度数组。如果 N 足够小，能使数组 A 的元素在 L1 cache 中都能命中，对于不同的 ofs，请预计循环的性能。 34

1.3 *硬件预取*。预取是一个有效利用内存接口的重要操作。x86 设计的硬件预取通常一次取满整内存页数据。试说明这可能对程序性能产生的负面效应。

1.4 *点积和预取*。考虑双精度浮点数的点积操作：

```
1 do i=1,N
2    s = s + A(i) * B(i)
3 enddo
```

N 非常大。CPU（时间周期为 1ns）能在一个周期内做一次读取（或者存储），一次乘法和一次加法（假设循环计数和分支不产生时间消耗）。存储总线的传输速率为 3.2GB/s。假设从存储读取一个 cache 行的延迟为 100 个 CPU 周期，一个 cache 行的长度为 4 个双精度浮点数。在以下情况下：

(a) 如果没有指令预取，循环的性能怎样？

(b) 假设 CPU 有预取的能力，为了使代码有效利用带宽（隐藏延迟），需要 CPU 能容忍预取多少条指令？

(c) 如果 cache 行的长度变为以前的 2 倍、4 倍，(b) 中算出的数值会怎样变化？ 35 ～ 36

(d) 如果我们假设指令预取能隐藏所有的延迟，循环的性能怎样？

第 2 章

Introduction to High Performance Computing for Scientists and Engineers

串行代码基本优化技术

在千核级并行计算机时代，有些观点认为编写高效串行代码在许多领域已经有些过时了。因为增加更多 CPU 以获得大规模并行能力要比投入大量精力优化串行代码简单得多。这似乎是一个合理的理论，5.3.8 节的论述中也体现了对这种观点的支持。然而，本书认为程序在单处理器上的性能优化毫无疑问是最重要的，如果通过一些简单的优化方法就可以实现两倍加速比，那么用户会更倾向于使用较少的 CPU。这样不仅可把宝贵的计算资源释放给其他用户或项目，而且还可以使投入大量资金购买的硬件获得更加有效的利用。因此，并行代码性能优化的第一步应该是在单处理器上的优化，使其运行速度加快。本章总结了串行代码剖析和优化的基本工具和策略。第 3 章会进行更深入的讨论，特别是数据传输优化的相关内容。

2.1 标量剖析

收集程序行为的相关信息，特别是程序对资源的使用信息，这个过程称为程序剖析。其中，在高性能计算方向最重要的“资源”是运行时间。因此，常见的剖析策略是评估程序中不同函数，甚至某几行关键代码的运行时间，并以此确定热点（hot spot，运行时间占主导地位的代码段）。这些热点随后会被剖析，并尽可能地进行优化。2.1.1 节分别介绍了基于函数和基于代码行的程序剖析方法。

然而，即使已经确定热点，但在很多情况（特别是这些函数或者代码块包含多行代码）下，造成性能瓶颈的原因却并不明确。这时，确定程序性能限制因素（如主存数据访问或流水线阻塞）的愿望就会非常迫切。如果数据访问是主要限制因素，则最直接的方法是确定导致程序性能降低的数据访存操作。解决这个问题的有效途径是使用硬件性能计数器，可提供当前系统使用的所有处理器信息，并提供芯片和系统内资源使用情况的深入分析。2.1.2 节对此会有详细讨论。

应该指出，在很多情况下，我们对串行代码的性能提升无能为力。因此，使用户能够确定优化工作的有效性至关重要。3.1 节提供了此类常见情况的指导。

2.1.1 基于函数和代码行的程序剖析

一般情况下，基于函数和代码行的剖析主要有两种技术：代码插入和采样。代码插入的工作原理是让编译器修改每个函数的调用，并插入代码以记录这些调用、调用者（或者完整调用栈）以及可能需要的时间信息。显然，这个技术会导致明显的额外开销（特别是当代码执行时间非常短时）。虽然代码插入技术会试图弥补这些开销，但仍然存在很多不确定性。相对这些额外开销，代码采样有着明显优势：程序在一定的时间间隔（如 10ms）内被周期性中断，并且记录程序计数器（或当前调用栈）信息。这个过程本质上是统计，代码运行的时间越长，产生的结果就越精确。结合目标代码相关信息，代码抽样还可以在源代码行甚至机

器代码行级别产生运行时间信息。由于效率的原因，限制了代码插入在很多函数或者代码块（只有一个入口和出口，代码块间又没有相互调用和跳转）上的使用。

1. 函数剖析

GNU binutils 包提供的 gprof 是使用最广泛的函数剖析工具。gprof 使用代码抽样和插入技术，收集函数剖析和调用图（也称为蝴蝶图）文件。为激活剖析功能，代码编译时必须指定合适的编译选项（许多现代编译器都能够产生与 gprof 兼容的指令：如 GCC，使用 -pg 编译选项）并运行一次。这将产生一个人工不可读、但可被 gprof 理解的输出文件：gmon.out。这个文件包含了程序中所有函数的执行时间以及它们的调用频次信息：

```
1   %   cumulative   self              self     total
2  time   seconds   seconds    calls  ms/call  ms/call  name
3  70.45     5.14     5.14 26074562     0.00     0.00  intersect
4  26.01     7.03     1.90  4000000     0.00     0.00  shade
5   3.72     7.30     0.27      100     2.71    73.03  calc_tile
```

每一行代表一个函数，各列解释如下：

%time：函数独立运行时间（不包含被该函数调用的其他函数的运行时间）占总运行时间的百分比。

cumulative seconds：所有函数的累积运行时间（包括自身）。

self seconds：函数的独立运行时间（单位：秒）。默认情况下，表单会根据这一列信息进行排序。

calls：函数被调用的次数。

self ms/call：函数每次调用的平均独立运行时间（单位：ms）。

total ms/call：函数每次调用的平均整体运行时间（包括被它调用的其他函数的运行时间，单位：ms）。

根据 gmon.out，上面实例的优化工作应该从 intersect() 函数开始，同时也要关注 shade() 函数的优化。函数的独立运行时间暗示了优化该函数所能获得的最大性能提升。例如，如果能将 shade() 函数的性能提高两倍，那么整个应用程序的运行时间为 7.3 − 0.95 = 6.35s，大约得到了 15% 的性能提升。

注意，程序剖析结果关键取决于编译器执行内联函数的能力。内联是一种使用函数体本身替代函数调用接口，以减少函数调用开销的优化方法（更加深入的讨论见 2.4.2 节）。如果编译器支持内联功能，那么剖析结果可能会被严重曲解（当热点函数被内联执行时，其运行时间会作为调用函数运行时间的一部分输出）。当内联函数对程序性能有明显影响，而编译器 / 剖析器不支持对内联函数的正确剖析时，需要禁止内联功能。当然，这样可能会对程序性能产生明显影响。

虽然剖析文件已经包含了大量信息。然而，它并没有说明当一个函数被几个不同函数调用时，对运行时间的贡献是怎么样的；这个函数又调用了哪几个子函数，当这些子函数依次调用时，对运行时间又贡献了多少。蝴蝶图（函数调用图）提供了这些信息。

```
1  index % time    self  children    called     name
2                  0.27    7.03     100/100        main [2]
3  [1]     99.9    0.27    7.03     100         calc_tile [1]
4                  1.90    5.14 4000000/4000000     shade [3]
5  -----------------------------------------------
6                                                   <spontaneous>
7  [2]     99.9    0.00    7.30                 main [2]
```

```
                0.27    7.03     100/100          calc_tile [1]
-----------------------------------------------
                                 5517592          shade [3]
                1.90    5.14 4000000/4000000     calc_tile [1]
[3]     96.2    1.90    5.14 4000000+5517592 shade [3]
                5.14    0.00 26074562/26074562     intersect [4]
                                 5517592          shade [3]
-----------------------------------------------
                5.14    0.00 26074562/26074562     shade [3]
[4]     70.2    5.14    0.00 26074562          intersect [4]
```

39 调用图的每一部分代表一个函数，最左边显示了该函数的运行索引。位于该索引之上的函数是当前函数的调用者函数（调用当前函数的函数），之下的是被调用者函数（被当前函数调用的函数）。同时，调用图也说明了函数的递归调用（见 shade() 函数）情况。调用图各个字段的含义如下：

%time 对应函数运行时间（包括被调用者函数的运行时间）占总运行时间的百分比。这应该等于剖析文件中该函数的“调用次数”与“每次调用整体运行时间”的乘积。

self 对应函数的独立运行时间（与剖析文件一样，不包括被调用者函数的运行时间）。对于调用者函数（被调用者函数）行，该列值说明了对应函数（被调用者函数）对其调用者函数（对应函数）的整体运行时间贡献。

children：对应函数的整体运行时间减去独立运行时间（对应函数的被调用者函数的总运行时间）。对于调用者函数行，其列值为对应函数的被调用者函数对其调用者函数运行的时间贡献。对于被调用者函数行，其列值为对应函数的被调用者函数的被调用者函数对该函数的运行时间贡献。

called：对应函数的调用次数（可分为递归调用次数 + 非递归调用次数，如 shade() 函数）。对于调用者函数行，其列值为该函数被调用者函数的调用次数。对于被调用者函数行，其列值为被调用者函数被该函数调用的次数。

目前有很多工具实现了蝴蝶图的可视化。借助可视化蝴蝶图，可浏览整个调用树并快速找到关“键路径”，即对整体运行时间起主导作用的函数序列：从根节点到某些叶子节点。

2. 基于行的剖析

如果待剖析程序中有大函数（代码行作为计量单位），这个函数的运行时间又在整体运行时间中占很大比例时，基于函数的剖析就显得力不从心了：

```
  %   cumulative   self              self     total
time   seconds   seconds    calls   s/call   s/call  name
73.21     13.47    13.47        1    13.47    18.40  MAIN__
 6.47     14.66     1.19 21993788     0.00     0.00  mvteil_
 6.36     15.83     1.17 51827551     0.00     0.00  ranl_
 6.25     16.98     1.15 35996244     0.00     0.00  gzahl_
```

如上表所示，Fortran 程序的 main 函数大约有 1700 行代码，运行时间占总体运行时间的 73%。如果使用简单方法不能确定这类函数的热点，就需要借助基于行的剖析工具。目前，有许多这种开源或者商业工具，可在不同层面上完成基于行的代码剖析工作。本节以开源工具 OProfile[T19] 为例说明基于行的程序剖析过程。需要说明的是，OProfile 也具有基
40 于函数的程序剖析功能，在某种程度上可代替 gprof。使用 OProfile 的唯一前提是程序必须进行可调式编译（通常通过添加编译选项 -g 实现），不需要其他特殊指令。然后启动后台剖析程序（通常由系统超级管理员启动），以监控整个计算机系统并收集正在运行的所有二进制文件信息。最后，用户可提取一个特定二进制文件的相关信息。除其他事项外，这个信息

可以看作带注释的源代码：每一行源代码都添加了相应的采样命中次数（第一列）和占全部程序采样的相对百分比（第二列）信息：

```
1                 :        DO 215 M=1,3
2   4292   0.9317 :           bremsdir(M) = bremsdir(M) + FH(M)*Z12
3   1462   0.3174 : 215     CONTINUE
4                 :
5    682   0.1481 :        U12 = U12 + GCL12 * Upot
6                 :
7                 :        DO 230 M=1,3
8   3348   0.7268 :           F(M,I)=F(M,I)+FH(M)*Z12
9   1497   0.3250 :           Fion(M)=Fion(M)+FH(M)*Z12
10   501   0.1088 :230      CONTINUE
```

然而，对这些数据的使用必须要特别小心。为使机器指令在内存中的地址能够和源代码行正确匹配，编译器生成的符号表必须保持一致性。如果开启高级优化，现代编译器会大量重组代码（比如，循环的融合与拆分、代码行的重新排列以及变量的优化消除等）。所以，实际执行的代码可能和原始代码差异很大。更进一步讲，由于现代处理器的流水线架构，查看特定源代码甚至机器指令在特定时刻的状态是不可能的。然而，采用基于循环的方法（取样自各个循环体）查看基于代码行的剖析数据还是相对安全的。如果对剖析信息有疑问，可对代码进行基于低级别优化的重新编译（禁用内联），这样可能会提供更加有意义的信息。

上述基于代码行的剖析数据可非常方便地放入表单中，以确定程序热点。如果某一行代码产生了较多的采样，所有样本的累积总和会在该行号上产生一个陡峭的斜坡（如图 2-1 所示）。

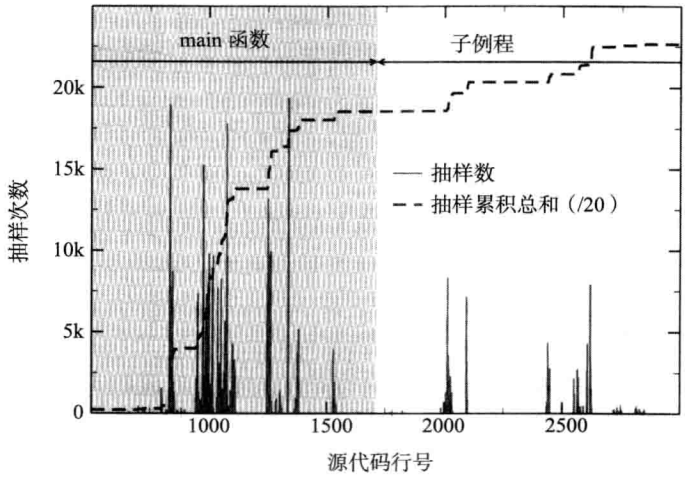


图 2-1 抽样直方图：横坐标为源代码行号；纵坐标为抽样次数；虚线为累积抽样总和。程序热点（占总运行时间的 50%）位于 main 函数（大约有超过 1700 行代码）的第 1000 行代码左右

2.1.2 硬件性能计数器

确定热点（如根据时钟周期）是程序性能剖析的第一步。然而，要确定程序性能低的原因或者确定限制程序性能的资源，仅有时钟周期信息是远远不够的。幸运的是，现代处理器都采用了少量性能计数器（通常远小于 10）。性能计数器是一种专用片上寄存器，当特定事

件发生时，其值会依次递增。在它可以监控的几百个事件中，下面几个信息是对程序性能剖析最有用的：

- 总线事务（即 cache 行数据传输）次数。一般情况下，会用“cache 未命中”来替
41 代总线事务。然而，由于预取机制（硬件或者软件实现）会干扰 cache 命中率的统计。使用“总线事务次数”统计内存总线传输的数据量是比较稳妥的方式。如果处理器实际使用的访存带宽接近理论峰值（更高或者接近 STREAM（访存带宽标准测试集）[W119,134] 能达到的访存带宽，具体见 3.1 节），再继续优化访存带宽利用率对性能提升就没有意义了。此外，“总线事务次数”还可用于验证一些理论模型（比如一个为应用程序评估数据传输量的模型）的正确性（3.1 节对这个模型进行了详细讨论）。
- 访存次数。结合总线事务次数可指导 cache 行的效利用。例如，如果从一个 cache 行中加载或者存储的 DP 数据小于 DP 字中 cache 行的大小，说明这是一个非连续访存。非连续访存一般会造成程序性能的降低。然而，我们必须要对数据的使用方式非常清楚。例如，由于某些原因，处理器流水线在绝大部分时间内阻塞或寄存器数据进行了大量算术运算，非连续访存可能就不是应用程序的性能瓶颈。
- 浮点数运算次数。这个基准的重要性经常被高估（正如第 3 章讨论，科学应用程序
42 的主要性能限制因素是数据传输）。如果每个 CPU 时钟周期所能进行的浮点数运算次数接近理论峰值（给定 CPU 的峰值性能，或者当乘法和加法操作不对称时相对较低的性能）时，基本的代码优化方法不太可能提升程序性能，这时可能要考虑算法上的改进。
- 分支预测失误。当 CPU 对条件分支预测错误时，该计数器递增。分支预测的时间消耗与具体的硬件架构相关，大约是数十个时钟周期（见 2.3.2 节的讨论）。科学应用程序往往是基于循环的，因此分支会被很好地预测。然而，“指针链接”和计算性分支增加了预测失误的可能性。
- 流水线阻塞。由于处理器流水线（见 1.2.3 节）不同阶段执行的操作间存在相互依赖，导致流水线某一阶段处于等待空闲状态，称为流水线阻塞，或者流水线气泡。通常情况下，流水线阻塞是不能避免的。例如，当访存带宽是程序性能的限制因素时，算术单元将耗费大量的时间等待操作数据。当有“太多”气泡时，确定优化方法是非常困难的。如果没有让具备执行条件的指令首先执行的机制，单纯依靠硬件是无法有效修正气泡现象的，因此阻塞周期分析在有序架构（如 Intel IA64）上非常重要。
- 执行指令条数。结合钟周期可成为判断如何有效利用具有多执行单元的超标量硬件架构（拥有多个执行单元）的准则。经验表明，即使有设计良好的流水线，并且有紧凑内循环代码，编译器生成代码的执行性能也很难达到每时钟周期 2 ～ 3 条指令。

使用硬件性能计数器进行程序剖析的方法基本有两种。我们常借助于工具快速了解应用程序的性能属性。这个工具不仅能够检测全部计数器信息，而且还可能计算生成许多派生计数器信息，如“每时钟周期指令数”和“每访存 cache 未命中数”。在这个工具中运行某些应用程序可能会产生类似的输出（这些例子都是在 SGI Altix 系统上根据 lipfpm 工具生成的输出进行编译的）：

```
1  CPU Cycles.....                                    8721026107
2  Retired Instructions.....                          21036052778
3  Average number of retired instructions per cycle.....  2.398151
4  L2 Misses.....                                     101822
5  Bus Memory Transactions.....                       54413
6  Average MB/s requested by L2.....                  2.241689
7  Average Bus Bandwidth (MB/s).....                  1.197943
8  Retired Loads.....                                 694058538
9  Retired Stores.....                                199529719
10 Retired FP Operations.....                         7134186664
11 Average MFLOP/s.....                               1225.702566
12 Full Pipe Bubbles in Main Pipe.....                3565110974
13 Percent stall/bubble cycles.....                   40.642963
```

43

通常使用的性能计数器数目非常少（一般为 2 ～ 4 个）。如需使用大量计数器（如上例）可能需要运行应用程序多次，或者剖析工具本身支持不同矩阵组的多路复用（如多个不同计数器组在一定时间间隔内可进行切换如 100ms）。后者会引入一个统计错误，这个错误必须密切关注（特别是涉及的计数非常小或者应用程序运行时间非常短的情况）。

上例中，每个时钟周期完成的指令数目（较多）、对 cache 和主存带宽需求（较小）、已完成的存取指令数目与 L2 cache 未命中数目的关系，都说明了这个硬件已经得到了很好的利用。流水线气泡占总 CPU 时钟周期的 40%，如果没有其他参照，很难说明这个值到底是高还是低。为了便于比较，下面是运行在相同硬件架构上的向量代码（采用较长的向量长度）的剖析结果：

```
1  CPU Cycles.....                                    28526301346
2  Retired Instructions.....                          15720706664
3  Average number of retired instructions per cycle.....  0.551095
4  L2 Misses.....                                     605101189
5  Bus Memory Transactions.....                       751366092
6  Average MB/s requested by L2.....                  4058.535901
7  Average Bus Bandwidth (MB/s).....                  5028.015243
8  Retired Loads.....                                 3756854692
9  Retired Stores.....                                2472009027
10 Retired FP Operations.....                         4800014764
11 Average MFLOP/s.....                               252.399428
12 Full Pipe Bubbles in Main Pipe.....                25550004147
13 Percent stall/bubble cycles.....                   89.566481
```

上例的带宽需求（较高）、每个时钟周期完成的指令数（较少）、存取指令数与 L2 cache 未命中数的关系，都表明这是一个访存受限应用。与前面的例子相比，阻塞指令所占比例增加了一倍以上。只有基于多个计数器、精心设计的阻塞指令周期分析，才能够解释这些气泡产生的原因。

尽管提供了一些重要信息，但收集“全局”硬件计数器信息在很多情况下还是过于简单。比如，如果将应用程序的性能剖析信息根据性能属性的较大差异（如 cache 受限与访存受限）分成许多子段，结合计数器信息可能会导致错误的结论。将计数器的增长限制到特定的代码段，可实现计数器剖析文件的分解并获得更具体的数据。最简单的工具是一个至少允许控制计数器的开启和禁用的 API 库。包含在 LIKWID[T20,W120] 包中的开源工具就可以做到这一点，更重要的是，这个工具和目前绝大多数 x86 架构的处理器兼容。

使用硬件性能计数器的更高级方式（OProfile 和 Intel VTune[T21] 等工具都支持）是将 44
采样机制应用于应用程序的代码行或函数事件上，这和基于代码行的程序剖析非常类似。与简单的定期获取指令指针或者调用栈的快照不同，这个方法对每个计数器（或者更精确些是

metric）定义了溢出值。当计数器增加到这个值时会产生一个中断，并进行 IP 或者调用栈采样。一个特定计数器的采样累积自然会发生在该计数器增加最频繁的地方。注意，以上方法不是基于整个程序而是基于函数，甚至是基于代码行的。然而，对硬件事件计数结果的正确理解需要相当多的经验。

2.1.3 手工代码插入

如果基于编译器的代码插入开销相对于应用程序太大，或者只想对程序的部分代码段进行剖析以获得相对简单的性能属性视图，手工代码插入是非常好的方法。程序员可在程序中插入计时函数如 gettimeofday()（见代码清单 1-2，wrapper 函数），或者插入剖析库加 PAPI（T22）（如果需要硬件计数器信息）。像在 2.1.1 和 2.1.2 节讨论的那样，许多剖析库都允许应用程序控制标准剖析机制的启动和停止 [T20]。在 C++ 代码中，如果由于模板和运算符重载技术的应用而使程序剖析文件变得非常混乱，这种方法将非常有用。

理解计时函数返回结果时要非常小心。当测试时间和计时器的最小计时单位是同一数量级（如用最小的时间间隔就可以完成）时，最容易发生对计时函数返回结果的错误解释。

2.2 优化常识

简单的代码修改经常会带来性能的显著提升。下面的章节总结了避免性能缺陷的几个最重要的“优化常识”。这些方法看似微不足道，但许多科学应用程序在应用这些方法后，性能都有了显著提升。

2.2.1 少做工作

重新组织代码以减少代码工作量，在很多情况下可显著提升性能。最常见的例子是循环
45 检测一组对象是否具有特定属性，任一对象具备该属性即可：

```
logical :: FLAG
FLAG = .false.
do i=1,N
  if(complex_func(A(i)) < THRESHOLD) then
    FLAG = .true.
  endif
enddo
```

如果 complex_func() 函数没有其他作用，FLAG 是唯一和循环外部通信的变量。这种情况下，FLAG 值一经改变，就立即退出循环可明显减少计算工作量（取决于条件判断变为 true 的概率）：

```
logical :: FLAG
FLAG = .false.
do i=1,N
  if(complex_func(A(i)) < THRESHOLD) then
    FLAG = .true.
    exit
  endif
enddo
```

2.2.2 避免耗时运算

算法的实现通常采用“步步为营”的策略。首先不做性能方面（因为在进行性能优化时，往往存在更改数值运算的风险）的考虑，将公式直接翻译成代码。第二步使用“便宜”运算

替代“昂贵”运算。三角函数和幂运算是“强”运算（“昂贵”运算）的典型代表。记住类似 x**2.0 的表达式是不会被编译器优化成 x*x 的，而指数（对数）的运算性能是很低的。避免“昂贵”运算的优化方法称为强度消减（strength reduction）。除上述简单情况外，“强”运算有时会关联一组有限的固定参数。下面是一个非平衡自旋系统仿真代码的例子：

```
integer :: iL,iR,iU,iO,iS,iN
double precision :: edelz,tt
...                                  ! load spin orientations
edelz = iL+iR+iU+iO+iS+iN            ! loop kernel
BF    = 0.5d0*(1.d0+TANH(edelz/tt))
```

程序的最后两行代码包含在一个循环中，并占用该应用程序几乎全部的运行时间。整型变量用于存储自旋取向（向上或者向下，对应值为 1 或者 −1），所以变量 edelz 的取值范围为 {−6,⋯,+6}。在整个应用程序中，tanh() 函数即便用硬件实现也是最耗时的操作（至少几十个时钟周期）。根据以上描述，可根据参数范围将该函数结果转存在一个数组中，循环内部完全消除对 tanh() 函数的调用。假设 tt 为定值，那么这个表格只需创建一次： 46

```
double precision, dimension(-6:6) :: tanh_table
integer :: iL,iR,iU,iO,iS,iN
double precision :: tt
...
do i=-6,6                                ! do this once
  tanh_table(i) = 0.5d0*(1.d0+TANH(dble(i)/tt))
enddo
...
BF = tanh_table(iL+iR+iU+iO+iS+iN) ! loop kernel
```

这个数组存储在访存性能非常高的 L1 cache 中。因此，相对于 tanh() 函数，该数组的查找时间可以忽略不计。由于该数组尺寸较小且被频繁调用，所以整个计算过程都会被存储在 L1 cache 中。

2.2.3 缩减工作集

程序代码在计算过程中或至少在整体运行时间中的内存使用量称为该代码的工作集。一般情况下，压缩工作集会提高 cache 命中率，对性能提升有正面影响。如何实现工作集压缩以及是否会带来性能提升，很大程度上取决于算法和它的实现。上例中，原始代码使用了 4 字节的整除存储自旋取向，工作集也因此远远大于所有处理器的 L2 cache。如果改变数组定义，使用 1 字节的整数来存储自旋取向。工作集会因此减小将近 4 倍，从而接近 cache 大小。

然而，并不是所有处理器都能有效处理“小”数据类型。如果处理器采用较大的字长，单字节类型数据通过移位和标记操作抽取，那么使用单字节整型数据的程序会非常低效。另一方面，如果可以采用 SIMD 指令，那么采用简单数据类型的程序就会非常高效（具体见 2.3.3 节）。

2.3 小方法，大改进

2.3.1 消除常用子表达式

消除常用子表达式经常被认为是编译器的任务。其基本思想是，在构造复杂表达式之前，预先计算其中被多次调用的子表达式，并将结果存储在临时变量中。在循环代码优化中，这个方法称为循环无关代码移出： 47

```
1 ! inefficient                          tmp=s+r*sin(x)
2 do i=1,N                              do i=1,N
3   A(i)=A(i)+s+r*sin(x)        ⟶        A(i)=A(i)+tmp
4 enddo                                 enddo
```

该优化方法可节省大量计算时间，特别是当子表达式中包含“强”操作（如 sin()）时。尽管子表达式消除可能会受其他代码的影响，编译器原则上能够检测到并进行有关优化工作。然而，如果这个操作还需要其他关联规则，那么编译器常常不会进行优化（2.4.4 节详细讨论了编译器优化和算术表达式重排序优化）。在实际应用中，手工完成这项工作是非常好的策略。

2.3.2 避免分支

“紧”循环（比如，循环体内部操作很少）常用的优化技术是软件流水（见 1.2.3）、循环展开和其他优化技术（见本节后面内容）。由于某些原因编译器自动优化失败或者优化不够充分，则会明显影响程序性能。如当循环体内部包含条件分支时，则很容易发生：

```
1  do j=1,N
2    do i=1,N
3      if(i.ge.j) then
4        sign=1.d0
5      else if(i.lt.j) then
6        sign=-1.d0
7      else
8        sign=0.d0
9      endif
10     C(j) = C(j) + sign * A(i,j) * B(i)
11   enddo
12 enddo
```

上面的矩阵向量乘实例，使用 if 表达式完成了对上三角矩阵（sign = 1）、下三角矩阵（sign = −1）以及对角线元素（sign = 0）的分别处理。一旦处理器遇到对应的条件分支，许多分支预测逻辑单元在计算结果可用之前就会采用基于统计的方法对该计算结果进行预测。一旦预测被证明是错误的（也称为分支预测失误或分支迷失），流水线将重新回到该分支位
48 置，这意味着时钟周期的浪费。此外，分支预测失败后，编译器也就不能继续进行循环展开或者 SIMD 向量化（见下一节内容）等后续优化。幸运的是，该循环嵌套可通过改进消除所有 if 表达式：

```
1  do j=1,N
2    do i=j+1,N
3      C(j) = C(j) + A(i,j) * B(i)
4    enddo
5  enddo
6  do j=1,N
7    do i=1,j-1
8      C(j) = C(j) - A(i,j) * B(i)
9    enddo
10 enddo
```

通过使用两个不同的内循环，条件分支被移出。要说明的是，这个循环嵌套还有更多的优化潜力。具体请参考第 3 章对数据访存操作优化的讨论。

2.3.3 使用 SIMD 指令集

尽管向量处理器也使用 SIMD 指令，微处理器对 SIMD 指令的使用也称为“向量化”，使用 SIMD 指令集更类似于现代向量化系统的多轨机制。一般而言，如果一条单一指令可

执行更多的操作，那么一个上下文中“可向量化”循环的性能可以更高。例如，尽可能使用“小”数据类型。从 DP 切换到 SP 可能会导致两倍的性能提升（具备 SIMD 能力的 x86 型 CPU[V104, V105]），而且还可以将更多的数据加载到 cache 中。

当然，选择 SIMD 指令，并不总能带来性能提升。如果应用程序性能严重受限于受访存带宽，不采用 SIMD 技术可弥补这一差距。使用 SIMD 指令，只会大大加快寄存器到寄存器操作的性能，但是会大大延长寄存器从内存子系统中获取新数据的时间。

图 1-8 描述的一条单精度加法指令可用在数组相加的循环中：

```
real, dimension(1:N) :: r, x, y
do i=1, N
  r(i) = x(i) + y(i)
enddo
```

在上例中，循环的每次迭代都是相互独立的，循环体内部没有条件分支，数据访存也为连续访存操作。然而，使用 SIMD 指令需要对循环代码（如上例中应用的）重新组织：多次迭代（与 SIMD 寄存器大小相等）间不允许有分支，能够像单一的“块”一样执行。即使没有 SIMD，这也是一个众所周知的优化方法——循环展开（详细讨论见 3.5 节）。因为循环的迭代次数一般不是寄存器大小的整数倍，所以余下的循环迭代还是会标量执行。忽略软件流水（参见 1.2.3 节）的伪代码如下：

49

```
! vectorized part
rest = mod(N,4)
do i=1,N-rest,4
  load R1 = [x(i),x(i+1),x(i+2),x(i+3)]
  load R2 = [y(i),y(i+1),y(i+2),y(i+3)]
  ! "packed" addition (4 SP flops)
  R3 = ADD(R1,R2)
  store [r(i),r(i+1),r(i+2),r(i+3)] = R3
enddo
! remainder loop
do i=N-rest+1,N
  r(i) = x(i) + y(i)
enddo
```

R1、R2、R3 都是 128 位的 SIMD 寄存器。理想情况下，上述操作由编译器自动实现。实际优化中，可使用编译指导语句，提示可向量化的代码（这些代码的向量化必须是安全并且有益的）。

在这个例子中，SIMD 读取和存储指令需要特别关注。操作对齐数据和非对齐数据的一些 SIMD 指令集是不同的。以 x86（Intel/AMD）架构为例，该架构拥有对齐和非对齐的“打包”SSE 读取和存储指令 [V107，O54]。如果将对齐的读取和存储指令应用于非对齐内存地址（不是 16 的倍数），则会抛出异常。如果编译器对应用到向量化循环中的数组的对齐情况一无所知，又不能对其影响和控制，尽管会带来性能损失，也一定要使用非对齐的读取和存储指令（或者使用一系列的标量化指令）。如果程序员不能肯定数据是对齐的，则强制编译器假设最优对齐是非常危险的。在某些架构上，对齐操作至关重要，需要尽一切努力保证读取和存储指令对齐到合适的地址边界。

迭代间存在真依赖的循环（见 1.2.3 节）是不能被 SIMD 以此种方式向量化的（然而也有转机，见习题 2.2）。

```
do i=2,N
  A(i)=s*A(i-1)
enddo
```

这里，编译器将会进行标量操作，也就意味着只使用SIMD寄存器的最低位（x86架构）。

值得注意的是，循环向量化没有固定的指导原则。一个（可能是最弱的）可能的定义是循环内所有算术运算的执行要充分利用SIMD寄存器（完全利用SIMD寄存器的宽度）。即便如此，仍然可以使用标量读取和存储指令，编译器也会认为这样的循环是向量化的。支持SSE的x86处理器，其寄存器的高64位和低64位可以独立使用。因此，上述循环的向量化
50 加法可看作为双精度加法：

```
1  rest = mod(N,2)
2  do i=1,N-rest,2
3     ! scalar loads
4     load R1.low = x(i)
5     load R1.high = x(i+1)
6     load R2.low = y(i)
7     load R2.high = y(i+1)
8     ! "packed" addition (2 DP flops)
9     R3 = ADD(R1,R2)
10    ! scalar stores
11    store r(i) = R3.low
12    store r(i+1) = R3.high
13 enddo
14 ! remainder "loop"
15 if(rest.eq.1) r(N) = x(N) + y(N)
```

上例中，如果操作数驻留在cache中，该版本并不能提供最佳性能。尽管算术运算（第9行）都是SIMD并行操作，而读取和存储却都是标量运算。由于缺乏完整的编译器报告，因此确定这样一个缺陷的唯一方法是手动检查生成的汇编代码。即使添加命令行选项或者源代码指导语句，编译器还是不能有效完成循环的向量化。那么，在使用汇编语言之前，一个“不得已而为之”的方法是使用编译器内部函数（compiler intrinsics）。内部函数和汇编指令非常相似，两者可被编译器进行1:1转换。由于编译器为内部函数提供了可映射到SIMD操作数的特殊数据类型，所以使用内部函数可使用户从追踪每个寄存器使用情况的繁重工作中解脱出来。内部函数不仅对向量化非常有用，而且在高级语言设计不能很好地映射到CPU某些特性的情况下，也非常有用。然而不幸的是，即使在同一个硬件架构上，编译器间的内部函数也互不兼容[V112]。

最后，必须强调的是，相对于真正的向量化处理器，RISC系统并不总能从向量化中受益。如果一个访存受限程序可使用寄存器或cache进行大量数据重用（见第3章例子），那么数据重用优化的潜在性能提升是非常大的，甚至可以放弃向量化优化。

2.4 编译器作用

通过利用编译器自动优化，高性能计算程序可以获得不同程度的性能改进。几乎每个现代编译器都可以在命令行上设置编译选项，以便对编译器优化目标程序进行细粒度控制。有些情况下可以简单地通过更换一个编译器来检查程序是否还存在性能提升空间。编译器需要
51 进行复杂的工作以将高级代码编写成的源程序编译为机器代码，同时要顾及到处理器内部资源。本章和下一章讨论的一些优化方法可以在某些简单情况下被编译器实现，但是涉及复杂的情况时就无法用编译器自动完成优化工作。始终要注意的一点是编译器可能足够聪明但是又可能非常蠢笨。在讨论编译器能力时，一条常用评价是“编译器应该能够识别”，这经常是一个错误的假设。

参考文献 [C91] 概述了几种当前常用 C/C++ 编译器的优化能力，并介绍了一些手工优化的技巧和指导。

2.4.1 通用优化选项

每种编译器都提供了一组标准优化选项（-O0，-O1，…），每个级别的优化都包括哪些优化方法并没有固定标准，需要参考相关手册。但是，所有编译器在 -O0 选项级别禁止大多数优化，因此这是调试和分析程序的正确方法，在更高级别的优化层次上，编译器进行检测和消除冗余变量、重排算术表达式等优化，因此调试器不能在代码和数据间提供一致的视图。

需要注意的是，某些问题只在高级优化层次上才出现。这可能是编译器的错误或缺陷，也可能是典型的错误，例如数组访问越界（读写索引超过了数组的界限），在 -O0 层次和 -O3 层次，数据被按照不同的方式组织，因此可能只在某个优化层次出错。这种错误很难被定位，有些情况下由于与优化器的冲突，甚至最常用的 printf 函数也不能帮助定位该类错误。

2.4.2 内联

内联通过插入被调用函数或子程序的全部代码来减少程序运行时的调用开销。例如每个被调用函数都会使用寄存器或者栈（具体要以来参数数量和调用方式决定）中资源来存储和传递函数参数，内联确实移除了将参数压入栈中的必要，并且使编译器可以在它认为需要的时候使用寄存器（而不是根据某些调用约定），从而减轻了寄存器压力，寄存器压力是指 CPU 没有足够的寄存器存储复杂计算或者循环体内的所有操作数（更多介绍见第 2.4.5 节）。最后，内联使得编译器可见的代码段增大并可以利用更多的在非内联情况下不可用的优化手段。程序员不应该依赖编译器优化内联代码，在性能关键段（例如循环体内），编译器看不 52
见“真正的”代码反而无法优化。

函数调用是否会影响性能依赖于调用次数，通常情况下，内联频繁调用的小程序将会获得最大的性能加速。在 C++ 代码中，内联是提高性能的基本方法，因为对简单数据类型的重载操作将会变为较小的函数，并且当内联函数返回一个对象时，临时复制可以被省略（更多 C++ 优化细节见 2.5 节）。

编译器通常有不同的编译选项来控制内联的自动优化程度，例如，在什么程度上（即代码行数量）一个子程序可以作为一个被内联的候选对象等。注意 C99 和 C++ inline 关键字只是对编译器的一个提示，应该检查编译器日志（如果可用，见 2.4.6 节）判断函数是否真正被内联。

相反，在多处内联一个函数可能会增大目标代码而导致过量使用 L1 指令高速缓存，例如如果循环体内的指令不能存储在 L1 指令高速缓存中，将会与数据传输一起竞争更高层次的高速缓存或者主内存，因此读取指令延迟就会增大。所以在设置内联标识时需要考虑正反两个方面的效用。

2.4.3 别名

通过程序语言规则和对源代码的理解，编译器需要提出确切的假设来限制自身产生优化机器代码的能力。典型的例子是在 C（以及 C++）语言中利用指针（或引用）形式参数：

```
1 void scale_shift(double *a, double *b, double s, int n) {
2   for(int i=1; i<n; ++i)
3     a[i] = s*b[i-1];
4 }
```

假设被指针 a 和 b 指向的内存区域不重叠，即 [a,a+n−1] 与 [b,b+n−1] 不重叠，那么该循环体中的读取和存储操作就可以按照任意顺序重排，编译器会应用它认为合适的任何软件流水方式或者展开循环并将读取和存储打包在一个程序块中，就像下面伪码（忽略其余循环）：

```
1 loop:
2   load R1 = b(i+1)
3   load R2 = b(i+2)
4   R1 = MULT(s,R1)
5   R2 = MULT(s,R2)
6   store a(i) = R1
7   store a(i+1) = R2
53
8   i = i + 2
9   branch -> loop
```

此时循环可以简单地进行 SIMD−向量化优化（见 2.3.3 节）。

然而，C 和 C++ 标准允许指针的别名，所以在优化时不能假设两个指针指向的区域不重叠，例如，如果 a==b，该循环变成了 1.2.3 节的“真依赖”的 Fortran 实例，读取和存储的执行顺序必须与程序中声明的一致：

```
1 loop:
2   load R1 = b(i+1)
3   R1 = MULT(s,R1)
4   store a(i) = R1
5   load R2 = b(i+2)
6   R2 = MULT(s,R2)
7   store a(i+1) = R2
8   i = i + 2
9   branch -> loop
```

编译器在缺少更多信息的情况下必须按照这种方式生成代码，SIMD 向量化优化也必须被排除，处理器硬件在某些限制条件下允许读取和存储的重排 [V104,V105]，但是必须保证程序语义。

在 Fortran 标准中禁止参数别名，这也是 Fortran 程序比相同的 C 程序快的主要原因之一。所有的 C/C++ 编译器都有控制别名程度的命令行选项（例如 Intel 编译器中的 -fno-fnalias 选项和 GCC 中的 -fargument-noalias，表明任何函数的两个指针实参都不指向同一区域）。如果编译器被告知没有参数别名，那么原则上就可以进行类似 Fortran 代码中的优化。但是应该注意，如果对优化过后的程序使用参数别名将会产生错误结果。

2.4.4 计算准确性

2.3.1 节已经提到，当要求满足结合律时，编译器有时禁止重排算术表达式，除非极高层次的优化开关被打开。原因在于浮点运算不满足结合律 [135]：如果 a、b 和 c 是有限精度的浮点数，则 (a+b)+c 一般不等于 a+(b+c)。如果必须保证优化前代码的正确性，就不能使用结合律规则，因此应由程序员确定是否可以手工重组算术表达式。现代编译器都有相关编译选项用来控制算术表达式的重组，即使是在高层次的优化开关被打开的情况下。

同时要注意非正规数，即比用非零最高有效位表示的最小值还小的浮点数，会极大地影响计算性能，如果可能并且轻微的正确性损失是可接受的，那么这些数应该在硬件计算中被
54 设为零。

2.4.5 寄存器优化

这是编译器优化（考虑使用寄存器）中最关键也是最复杂的任务之一，编译器试图将寄存器分配给使用最频繁的操作数并将这些操作数尽可能长的保留在寄存器中（如果这样做安全）。例如，如果一个变量的地址被访问，该变量的值很可能被改变（通过地址操作由程序其他部分改变），这种情况下编译器要决定是否将该变量写回内存中。

内联（见 2.4.2 节）可以减轻寄存器优化负担，因为编译器可以将本应该在函数调用前写入内存并随后再读出的变量保存在寄存器中。相反，优化拥有大量变量和算术表达式的循环体（可能出现在内联优化后）对于编译器来说非常困难，因为编译器要保持的操作数数目过大而不能在一次迭代中同时存储所有操作数。前面提到过，处理器中整数寄存器和浮点数寄存器的数量通常是有限的。目前典型的数目为 8 ～ 128，如果寄存器数量不够，将会带来寄存器溢出，即将寄存器变量写回内存以供后续使用。如果程序的性能瓶颈是算术操作，那么寄存器溢出将会极大地降低性能。这种情况下可以划分一个大循环为多个循环来减轻寄存器使用压力。

一些处理器具备硬件支持可以处理寄存器溢出，例如 Intel Itanium2 处理器具有硬件性能计数器可以直接检测存储器溢出。

2.4.6 利用编译日志

前面几节指出编译器在编写高效程序中的关键作用。可以简单地操作以阻止编译器获得重要信息从而限制优化的级别和种类。为了更好地利用编译器的智能，需要编译器允许产生 55
注释源代码表或者编译日志来表示本次编译过程都使用了哪些优化方法。代码清单 2-1 展示了一个 MIPS R14000 处理器上（已过时）编译注释的例子，即代码清单 1-1 中的标准三元组向量程序。该处理器是一个四路超标量处理器，在一个时钟周期内可以同时执行一个读取或存储操作，两个整数操作，一个浮点加和一个浮点乘操作（后两个操作通过一条融合的乘 - 加指令“ madd”实现）。假设所有的数据都存储在最高层的高速缓存中，编译器可以计算程序一次循环迭代所需的最小计算周期数（第 3 行）。4 ～ 9 行展示了处理器峰值，即每类指令的最大吞吐量。

代码清单 2-1 流水化三元组软件的编译日表。其中“峰值”(peak) 指该体系结构 (MIPS R14000) 上各种操作类型的最大执行速率

```
1  #<swps> 16383 estimated iterations before pipelining
2  #<swps>     4 unrollings before pipelining
3  #<swps>    20 cycles per 4 iterations
4  #<swps>     8 flops         ( 20% of peak) (madds count as 2)
5  #<swps>     4 flops         ( 10% of peak) (madds count as 1)
6  #<swps>     4 madds         ( 20% of peak)
7  #<swps>    16 mem refs      ( 80% of peak)
8  #<swps>     5 integer ops   ( 12% of peak)
9  #<swps>    25 instructions  ( 31% of peak)
10 #<swps>     2 short trip threshold
11 #<swps>    13 integer registers used.
12 #<swps>    17 float registers used.
```

除此之外，寄存器利用和寄存器溢出（第 11 行和第 12 行），循环展开因子和软件流水因子（第 2 行，见 1.2.3 节和 3.5 节）、SIMD 指令的使用（见 2.3.3 节）以及循环次数的编译器假定（第 1 行）都在表中展示，可以用来判断生成的机器代码的质量。然而，并不是所有

编译器都有产生如此丰富的代码标注特性的能力，剩下的工作需要程序员完成。

也有人工检查汇编代码的编译选项。所有编译器提供命令行选项以输出汇编而不是可连接文件。然而，将该汇编文件与源代码进行对应并分析指令序列的效率需要很多经验[O55]。毕竟这是摒弃编写汇编语言代码的原因。

2.5 C++ 优化

目前，有大量关于如何编写高效 C++ 代码的文献 [C92,C93, C94, C95]。我们的目标不是取代它们。所以我们特意忽略了引用计数、写时复制、智能指针等关键技术。本节以循环代码为例，根据我们的经验指出 C++ 编程中经常存在的性能错误和误解。

C++ 编程存在着一个根深蒂固的假象：编译器应该能够识别高级 C++ 程序包含的所有抽象和代码混淆。首先，C++ 是一门支持复杂管理的高级编程语言，且自身特征明显（如运算符重载、面向对象、自动构建 / 销毁等）。然而，这些特征绝大多数都不适合编写高效的低层次代码。

2.5.1 临时变量

C++ 具有一个“隐式”的编程风格：自动机制为程序员隐藏了 C++ 编程的复杂性。然
56 而，在表达式含有运算符重载链时，经常会出现一个问题。例如，假设有一个表示三维向量的类 vec3d，该类实现了算术运算符重载以支持更有表现力的编码：

```
class vec3d {
  double x,y,z;
  friend vec3d operator*(double, const vec3d&);
public:
  vec3d(double _x=0.0, double _y=0.0, double _z=0.0) : // 4 ctors
        x(_x),y(_y),z(_z) {}
  vec3d(const vec3d &other);
  vec3d operator=(const vec3d &other);
  vec3d operator+(const vec3d &other) {
    vec3d tmp;
    tmp.x = x + other.x;
    tmp.y = y + other.y;
    tmp.z = z + other.z;
  }
  vec3d operator*(const vec3d &other);
  ...
};

vec3d operator*(double s, const vec3d& v) {
  vec3d tmp(s*v.x,s*v,y,s*v.z);
}
```

这里我们只给出了 vect3d::operator+ 和友元函数 vect3d::operator*(与一个标量相乘) 的实现。其他有用函数都以类似的方式定义。注意这里只给出了复制构造函数和赋值运算符的函数声明，这两个函数都是隐式定义的。因为对于这个类来说，默认的复制和赋值操作已经足够了。

下面代码段作为一个启发式的例子，说明了当类被调用时，背后究竟发生了什么：

```
vec3d a,b(2,2),c(3);
double x=1.0,y=2.0;

a = x*b + y*c;
```

在这个实例中，会按顺序逐步发生如下操作：

1）调用构造函数实例化 a、b、c、d 对象（根据参数调用相应的构造函数）。

2）调用 operator*(x, b) 函数。

3）调用构造函数初始化 operator*(double s, const vec3d& v) 中的 tmp 变量（这里，我们没有使用默认构造函数，而是选择了更加高效的接受三个参数的构造函数）。

4）因为在函数 operator*(double, const vec3d&) 返回时，tmp 变量会被销毁。所以 vect3d 的复制构造函数被调用，以创建一个临时变量存储 tmp 结果，并将其作为“加”运算的第一个参数。 57

5）调用 operator*(y, c)。

6）调用构造函数初始化 operator*(double s, const vec3d& v) 中的 tmp 变量。

7）因为函数 operator*(double, const vec3d&) 返回时，tmp 变量会被销毁。所以 vect3d 的复制构造函数被调用，以创建一个临时变量存储 tmp 结果，并将其作为“加”运算的第二个参数。

8）第一个临时对象调用 vec3d::operator+(const vec3d&) 函数，第二个临时对象作为其参数。

9）调用默认构造函数，初始化 vec3d::operator+ 函数中的 tmp 对象。

10）调用 vec3d 的复制构造函数，完成运算结果的临时拷贝操作。

11）调用 vect3d 的赋值运算符，临时拷贝作为其参数。

尽管编译器可能会使用所谓的返回值优化消除本地变量 tmp[C92]，而直接使用隐式临时变量而不是 tmp。然而，完成一个看起来如此简单的表达式所要执行的代码数量是如此的复杂（使用调试工具可查看相关详细信息）。对此，一个直接的优化策略是使用复合计算或赋值运算符（以牺牲可读性为代价），如 +=：

```
1 a  = y*c;
2 a += x*b;
```

这里仍然需要两个临时变量将 operator*(double, const vec3d&) 函数的结果返回到主函数。但是它们直接被赋值运算符和 vec3d::operator+= 应用，这样就不需要第三个临时变量。该优点在较长操作链中体现得更加明显。

然而，即使处理临时变量（比如，调用复制构造函数）消耗了大量的计算时间，标准函数剖析文件（见 2.1.1 节内容）也不一定能将此清楚显示。C++ 编译器非常擅长函数内联，由此会引发许多“神奇”的事情：比如一个包含复杂表达式函数的独立运行时间。在这种情况下，禁用函数内联功能（虽然一般情况下不支持这么做）可能会得到更多的信息。然而这样会严重干扰剖析结果。

尽管会积极使用内联功能，编译器也不太可能生成“最优”代码。其生成的代码大致上是这样的： 58

```
1 a.x = x*b.x + y*c.x;
2 a.y = x*b.y + y*c.y;
3 a.z = x*b.z + y*c.z;
```

表达式模板（expression template）[C96,C97] 是一种先进的编程技术，应该可以解决很多临时变量引发的性能问题。实际上，通过高级表达式它也会生成这样的代码。

应该明确的是，C++ 内联功能不是为了生成最优代码，而是要弥补因语言规范导致的最严重的性能损失。受内存带宽甚至 cache 带宽或者算术吞吐量限制的循环代码，最好用 C 或者 Fortran 编写（2.5.3 节将进行详细讨论）。

2.5.2 动态内存管理

C++ 代码中另一个常见的性能瓶颈是频繁的内存分配和释放。上节讨论的 vec3d 类，由于没有涉及动态内存，所以不存在大量内存分配（释放）的问题。如果我们选择一个类似于 vec3d 但所占内存空间可变的类，其构造函数和析构函数会分别调用 malloc() 和 free() 函数。因此，临时变量对性能的影响会更加严重。而标准库函数并没进行最佳性能优化，因此会严重损害程序的整体性能。这就是 C++ 程序员竭尽全力试图减小内存分配和释放对性能影响的原因。

上节讨论的是避免临时变量而采取的其中一个关键措施。除此之外，还有另外两个有效策略：延迟构造和静态构造。这两个策略看起来是对立的，但它们都是有用的策略。

1. 延迟构造

将 C++ 作为“第二语言”的 C 程序员一般会在函数的开始就声明所有变量，而不是需要时才声明。前者是 C 语言所需的，只要使用的是基本数据类型就不存在性能问题。然而，要尽量避免“昂贵”的构造函数如下所述：

```
1 void f(double threshold, int length) {
2   std::vector<double> v(length);
3   if(rand() > threshold*RAND_MAX) {
4     v = obtain_data(length);
5     std::sort(v.begin(), v.end());
6     process_data(v);
7   }
8 }
```

尽管使用变量 v 的概率可能会非常低（依赖于 threshold），但第 2 行代码还是无条件对
59 变量 v 进行了声明。一个更好的方案是在需要它时再声明：

```
1 void f(double threshold, int length) {
2   if(rand() > threshold*RAND_MAX) {
3     std::vector<double> v(obtain_data(length));
4     std::sort(v.begin(), v.end());
5     process_data(v);
6   }
7 }
```

这样编写代码的另一个好处是：可以直接调用 std::vector<>（第 3 行）的复制构造函数。而不像之前那样：首先调用构造函数（带 int 型参数），然后再调用赋值运算符。

2. 静态构造

如果对象的使用非常频繁，将其构造放在循环或者代码块的外面，或者声明为 static 变量，其性能可能会比延迟构造要高得多。如上例，如果数组的长度是个常量且 threshold 值接近 1，那么静态分配可使构造开销忽略不计（因为只构造一次）。

```
1 const int length=1000;
2
3 void f(double threshold) {
4   static std::vector<double> v(length);
5   if(rand() > threshold*RAND_MAX) {
6     v = obtain_data(length);
7     std::sort(v.begin(), v.end());
8     process_data(v);
9   }
10 }
```

向量对象只实例化一次（第 4 行），并且没有后续分配开销。然而，如果向量长度可变，

那么内存不得不重新分配，从而产生了和正常构造相同的开销（见习题 2.4）。一般情况下，如果赋值操作比内存分配快（平均值），则静态分配性能会更高。

并行程序中存放在共享内存的静态数据要特别关注，详细内容见 6.1.4 节。

2.5.3 循环与迭代器

循环（或者循环嵌套）在科学应用程序的运行时中占主导地位。编译器对这些循环的优化能力是获得高性能代码的关键。运算符重载可能会对编程带来很多便利，但不利于循环优化。下面的例子中，模板函数 sprod<>() 实现了两个向量的内积。

```
1 using namespace std;
2
3 template<class T> T sprod(const vector<T> &a, const vector<T> &b) {
4   T result=T(0);
5   int s = a.size();
6   for(int i=0; i<s; ++i)    // not SIMD vectorized
7     result += a[i] * b[i];
8   return result;
9 }
```

60

在代码第 7 行，const T& vector<T>::operator[] 被调用了两次，分别获得向量 a 和 b 的相应分量。STL 定义这个操作的方式如下（改编自 GNU ISO C++ 库代码）：

```
1 const T& operator[](size_t __n) const
2     { return *(this->_M_impl._M_start + __n); }
```

尽管代码看起来足够简单，可以有效内联。然而，目前编译器拒绝为上例中的求和循环进行 SIMD 向量优化。一个单一的抽象层（索引运算符的重载）就可以阻止最优循环代码的生成（我们甚至都没有提及第 3 章中列举的更复杂、更高层次的循环转换）。然而，当使用迭代器进行数组元素访问时，向量优化将不是问题：

```
1 template<class T> T sprod(const vector<T> &a, const vector<T> &b) {
2   typename vector<T>::const_iterator ia=a.begin(),ib=b.begin();
3   T result=T(0);
4   int s = a.size();
5   for(int i=0; i<s; ++i)    // SIMD vectorized
6     result += ia[i] * ib[i];
7   return result;
8 }
```

因为 vector<T>::const_iterator 是 const T*，所以编译器认为这是正常的 C 代码。在 C++ 编程中，使用迭代器进行数据访问是一个有效的优化方法。如果有可能，低层次循环代码甚至应该驻留在单独的编译单元上（用 C 或者 Fortran 编写），并且迭代器可作为指针参数传递过去。保证尽量不干扰编译器对高级 C++ 代码的编译。

std::vector<> 模板（最常用的容器）是一个特例，因为它的迭代器实现和标准（C）指针一样。而越复杂的容器则有更复杂的迭代器类，可能不太容易转换为原始指针。这种情况下，可使用包含多个类 vector<> 组件的“分段”结构表示数据（矩阵就是一个典型例子）。分段迭代器的使用还可实现快速低级别算法。详细信息见 [C99, C100]。

61

习题

2.1 *分支的危险。*考虑下面的基准代码：

```
1 do i=1,N
2   if(C(i)<0.d0) then
```

```
3      A(i) = B(i) - C(i) * D(i)
4    else
5      A(i) = B(i) + C(i) * D(i)
6    endif
7  enddo
```

相对于标准向量三元组，请分析条件分支在下面情况下对性能的影响。数组 C 的值初始化为：a）正值、b）负值、c）[−1,1] 的任意值，并分别存放在 L1 cache、L2 cache 和内存中。

2.2 *递归能 SIMD 向量化吗？* 在 1.2.3 节我们通过下面循环代码学习了循环体依赖对流水线的影响：

```
1 start=max(1,1-offset)
2 end=min(N,N-offset)
3 do i=start,end
4   A(i)=s*A(i+offset)
5 enddo
```

如果 A 是一个单精度浮点数数组，当 offset 取何值时，该循环可被 SIMD 向量化（如图 1-8 所示）。

2.3 *栈上的延迟构造。* 在 2.5.2 节的延迟构造例子中，如果我们使用标准 C 的 double 数组，而不是使用 std::vector<double>。数组被声明时的区别在哪里？

2.4 *快速赋值。* 在 2.5.2 节静态构造例子中，我们指出静态对象 std::vector<> 只有在长度为常量时才具有优势。如果长度可变，赋值操作会导致内存的重新分配，真的是这样吗？

62

数据访存优化

在高性能计算中，访存是最重要的性能限制因素。如前所述，微处理器的理论峰值性能和访存带宽存在固有的“不平衡性”。因为很多科学和工程应用程序由需要大量数据传输的基于循环的代码构成，所以相对较低的内存（甚至是硬盘）访存带宽，就会导致片上资源的低效利用和程序性能的降低。

图 3-1 综合显示了现代并行计算机系统的数据通路构成，以及在不同层次上的带宽和延迟范围。执行计算任务的功能部件位于该层次结构的顶部。在这些不同层次的数据通路中，访存带宽最大有 3 ～ 4 个数量级的差异，访存延迟最高也有 8 个数量级的差异。为了获得计算操作数，数据传输需要到达的数据通路层次越深，对性能的影响就越小。因此，任何优化首先要考虑减少在性能低的数据通路上的数据传输。当这不能实现时，至少要让数据传输尽可能高效。

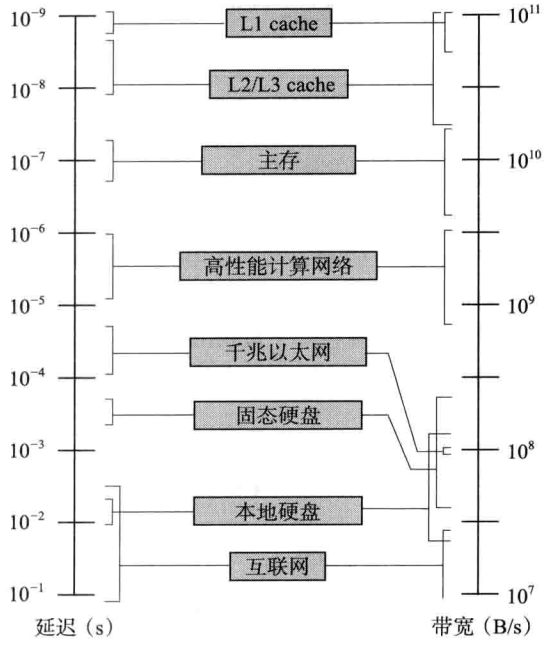


图 3-1　计算机系统中不同设备间数据传输延迟和带宽的典型值。图中忽略了寄存器，这是因为寄存器带宽通常与计算核心的计算能力相匹配，而且其延迟是流水线执行的一部分

3.1　平衡分析与 lightspeed 评估

3.1.1　基于带宽的性能建模

许多程序员都会竭尽全力提高程序性能。为了确定这些努力是否有效或者确定目标程序是否已经得到了充分优化，程序员会使用简单的经验法则来评估这些基于循环且带宽访存受限的代码的理论性能。这里引入一个核心概念：平衡（balance）。例如，一个处理器芯片的“机器平衡” B_m 等于可能的访存带宽（GWord/s，千兆字 / 秒）与峰值性能（GFlop/s，千兆浮点运算次数 / 秒）的比值：

$$B_m = \frac{\text{memory 访存带宽 [GWord/s]}}{\text{peak 峰值性能 [GFlop/s]}} = \frac{b_{max}}{P_{max}} \tag{3-1}$$

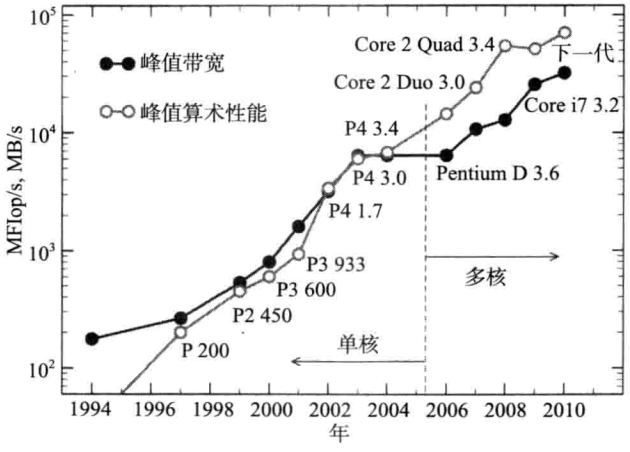
尽管“访存带宽”通常对真正访存受限的代码有用，但这个术语可被 cache 带宽或者网

63 络带宽代替。假定访存延迟被预取和软件流水这类技术隐藏，例如，一个时钟频率为 3.0 GHz 的双核芯片，每核每个时钟周期最多可执行 4 个浮点数计算，访存带宽为 10.6GB/s，则这个处理器的机器平衡值为 0.055W/F。在本书进行写作的时候，B_m 值一般位于 0.03W/F（标准的基于 cache 的微处理器）～ 0.5W/F（高端行向量处理器）之间。由于不断增长的 DRAM 差距和不断增加的处理器核数，未来标准架构的机器平衡值可能会降低。表 3-1 显示了不同数据传输路径的平衡值。

表 3-1 不同传输路径限制的典型的平衡值。在网络和磁盘的平衡值计算中，以典型的双路计算节点的峰值性能为基准

数据路径	平衡值 [W/F]
cache	0.5 ～ 1.0
机器（内存）	0.03 ～ 0.5
高速互联	0.001 ～ 0.02
千兆以太网互联	0.0001 ～ 0.0007
磁盘（或磁盘子系统）	0.0001 ～ 0.01

图 3-2 收集了 1994 ～ 2010 年间，英特尔处理器的峰值性能和访存带宽信息（选取每年时钟性能最高的桌面处理器作为代表）。尽管 2005 年之前峰值性能的增长速度要快于访存带宽，但第一款双核芯片（Pentium D）的推出才真正大幅增大了 DRAM 差距。虽然 Core i7 为访存带宽赢回了一些颜面，但长期的发展趋势显然不会受到这个例外的影响。



64～65 图 3-2 1994 年起，英特尔处理器的最大算术运算性能（空心圆）和理论峰值访存带宽（实心圆）的发展。选取每年时钟频率最高的处理器作为代表（数据收集自 JanTreibig）

为了量化运行在具有特定平衡值机器上的代码需求，我们进一步定义了循环的代码平衡值：

$$B_c = \frac{\text{数据传输量 [Word]}}{\text{浮点数运算点数 [Flop]}} \tag{3-2}$$

“数据传输量”表示通过数据通路（性能限制因素）传输的所有数据量（单位为字），这个术语在一定程度上依赖于硬件（见 3.3 节实例）。代码平衡值的倒数称为计算密度（computational intensity）。现在我们可以计算出代码平衡值为 B_c 的代码在机器平衡值为 B_m

的设备上所能达到的峰值性能的最大期望：

$$l = \min\left(1, \frac{B_m}{B_c}\right) \tag{3-3}$$

我们将这个小数称为循环的“lightspeed”。则所能达到的峰值性能（GFlop/s）为：

$$P = lP_{max} = \min\left(P_{max}, \frac{b_{max}}{B_c}\right) \tag{3-4}$$

如果 $l \approx 1$，则代码性能的限制因素将在 CPU 或其他地方，访存带宽不再是代码性能的限制因素。值得注意的是这个简单的性能模型是基于以下几点重要假设：

- 循环代码应以最佳方式使用所有的算术单元（乘法和加法）。如没有，则必须引入一个校正因子来反映有效峰值性能和绝对峰值性能的比值（例如，如果代码只使用了加法单元，则有效峰值性能为绝对峰值性能的一半）。当使用的内核数量少于芯片上的内核总数时，也要有类似的考虑。
- 虽然循环代码基于双精度浮点数运算。然而我们可以非常容易地得到更加适合其他代码的相似术语（例如，32 位字每整数算术运算指令）。
- 数据传输和运算完美重叠。
- 最慢的数据通路决定了循环代码的性能。所有更快的数据通路都被认为是无限快的。
- 计算系统面向吞吐量，即忽略延迟对性能的影响。
- 完全有可能实现对内存带宽的利用达到饱和，从而使机器平衡值的计算达到最大值。近年来的多核架构设计中如果只使用其中一部分核，则对内存接口的使用是低效的。这使得性能预测更加困难，因为这需要一个单独的“有效”机器平衡值，这个值不仅是简单地使用核数与总核数的比值。3.1.2 节和习题 3.1 对这点会有更加详细的讨论。 66

虽然平衡模型经常很有用，足够评估循环代码的性能，并能对性能优化提供指导（特别是同可视化结合起来，比如 roofline 模型 [M42]）。我们必须强调的是确实存在更加先进的性能模型策略，参见文献 [L76, M43, M41]。

我们以 1.3 节介绍的标准向量基准代码为例：

```
1 do i=1,N
2   A(i) = B(i) + C(i) * D(i)
3 enddo
```

该内核每个迭代循环执行两次浮点数运算、三次数据读取指令（读取数据 B(i)、C(i) 和 D(i)）和一次数据存储指令（存储 A(i)）。代码平衡值为 $B_c = (3+1)/2 = 2$。如果该代码运行在机器平衡值 $B_m = 0.1$ 的 CPU 上，那么我们可以预期 lightspeed 比为 0.05，即峰值的 5%。

基于 cache 的标准微处理器，其最外层 cache 通常会具有回写策略。如 1.3 节所述，如果延迟写或者行 0 未被使用时，当一次写操作未命中时，为保证 cache – 存储器的一致性，需要一次 cache 行的写分配。在这样的条件下，计算代码平衡值时，数组 A 的存储操作必须计算两次，这样我们计算出最终的 lightspeed 为 $l_{wa} = 0.04$。

3.1.2 STREAM 基准测试

McCalpin STREAM 是用来评估处理器或者系统内存接口性能的基准测试，包含四个简单的合成内核循环代码。表 3-2 列出了它们的操作以及各自的代码平衡值。有效内存带宽（GB/s）通常作为其性能提供报告。注意不要将 STREAM TRIAD 内核与上一节讨论的三元组向量操作混淆，它包含一个额外的数据读取操作。

表 3-2 STREAM 基准测试内核每次循环需要传输的数据量（第三列）和浮点数运算量（第四列）。括号中的数字将写入操作也计算在内

类型	内核	DP Word	Flop	B_c
COPY	A(:)=B(:)	2(3)	0	N/A
SCALE	A(:)=S*B(:)	2(3)	1	2.0(3.0)
ADD	A(:)=B(:)+C(:)	2(4)	1	3.0(4.0)
TRIAD	A(:)=B(:)+S*C(:)	2(4)	2	1.5(2.0)

STREAM 基准测试存在串行和 OpenMP 并行两个版本（参见第 6 章）。这两个版本通常都运行在足够大的数据集上，以确保内核性能都是访存受限的。因此，内存带宽测试仅依赖
67 于数据读取和存储操作的次数，COPY 和 SCALE（ADD 和 TRIAD 同样适用）内核代码的性能结果往往类似。需要搞清楚的是，STREAM 并不是仅通过表 3-2 列出的循环内核定义的，还包含 Fortran 源代码（也有可用的 C 语言版本）。这一点非常重要，因为经过优化的编译器会识别出 STREAM 源代码，从而使用手工优化的机器代码代替内核代码。因此，STREAM 的性能结果可以真实地反映出硬件的实际性能。在 STREAM 网站上可以找到许多针对过去和现在的计算机系统的基准测试代码 [W119]。

遗憾的是，STREAM 同三元组向量代码一样，达不到平衡分析所预测的性能水平，特别是在商业（基于 PC 的）硬件上。其中的原因是多方面的，这里不进行详细讨论，主要原因如下：

- 在读和写两个方向上往往不能同时达到峰值带宽。例如，存在这种情况，内存读和内存写的峰值带宽的比例为 2∶1，在这种情况下，写操作并不能充分利用内存带宽。
- 协议开销（参见 4.2.1 节）、芯片缺陷、内存芯片的错误校正，以及高延迟（不能完全被预取隐藏）都会降低有效带宽。
- 处理器芯片上的数据通路（如 L1 cache 和寄存器间的数据通路），可以是单向的。如果代码在读和写操作间没有取得很好的平衡，那么在一个方向上的内存带宽可能不会得到充分利用。这是在 cache 上进行平衡分析时应该考虑的内容。

然而，STREAM 结果所标记的最大内存带宽依然是正确的。没有具有类似特征（读写操作的数量）的实际应用代码可以表现得更好。这样，在 lightspeed 计算中应该引用 STREAM 所测得的带宽 b_S，而不是硬件的理论值。式（3-4）可修正为：

$$P = \min\left(P_{\max}, \frac{b_S}{B_c}\right) \tag{3-5}$$

如果一个实际应用程序能够达到基于 STREAM 预测带宽的较大比值（如 80%），往往意味着该程序没有继续改进内存接口利用率的余地。当然，这并不意味着没有进一步优化的空间（如下面章节的讨论）。

我们选择一个使用英特尔 Xeon 5160 处理器（参见图 4-4）的系统作为测试平台。该处理器的每核理论内存带宽为 $b_{\max}$ = 10.66 GB/s，峰值性能为 $P_{\max}$ = 12 GFlop/s（3.0GHz，每个时钟周期可执行 4 次浮点数运算），则单核机器平衡值为 B_m = 0.111 W/F（如果两个核都运行访存受限代码，则该值需除以 2。但是目前我们假定每个系统通路只运行一个线程）。

68 表 3-3 显示了 STREAM 在这个平台上，有无写分配的比较测试结果。由于基准测试代码并没有考虑这个细节，所以当存在写分配操作时，测试带宽和实际内存传输不同。如表 3-3 所示，实测性能和理论峰值性能的差距非常明显，在这个平台上几乎不可能超过峰值带宽的 40%。ADD 与 TRIAD（两次读取操作而不是一次）内核的效率特别低。如果这个结果被用来做循环代码的平衡分析，COPY 和 SCALE 应该运行在加载 / 存储（写入）平衡的情

况下。3.3 节将会讨论一个具体实例。

表 3-3 英特尔 Xeon 5160 处理器有无写分配操作的单线程 STREAM 带宽（GB/s）的比较测试结果。写分配可通过使用延迟存储指令避免

	写分配			无写分配	
类型	实测	实际	b_S/b_{max}	实测	b_S/b_{max}
COPY	2698	4047	0.38	4089	0.38
SCALE	2695	4043	0.38	4106	0.39
ADD	2772	3696	0.35	3735	0.35
TRIAD	2879	3839	0.36	3786	0.36

3.2 存储顺序

多维数组、矩阵或者类矩阵结构（最重要），在科学计算中无处不在。数据访问是一个关键的问题：标准计算机所固有的一维、基于 cache 行的内存布局和任何多维数据结构间的映射必须与数据读取与存储的顺序相匹配，这样才能充分利用空间和时间局部性。一维数组的非连续访存会减少空间局部性，从而导致访存带宽利用率的低效（见习题 3.1）。当处理多维数组时，这些访存模式可以很自然地产生。

间隔 *N* 访存

```
1 do i=1,N
2   do j=1,N
3     A(i,j) = i*j
4   enddo
5 enddo
```

间隔 1 访存

```
for(i=0; i<N; ++i) {
  for(j=0; j<N; ++j) {
    a[i][j] = i*j;
  }
}
```

上面的 Fortran 和 C 代码示例执行相同的任务，数组的二维索引也都位于内层循环，但 69
是两者的数据访存模式完全不同：Fortran 代码的内存地址的访问跨度为 N*sizeof(double)，C 代码的内存地址的访问跨度是最优的（访存跨度为 1）。这是因为对于多维矩阵在 C 语言中是按行存储（见图 3-3），而在 Fortran 语言中是按列存储（见图 3-4）。这在进行数据访问优化时必须要牢记：当内层循环变量作为多维数组索引时，应该保证跨度为 1 的数据访问模式。3.4 节将详细讨论如何实现。

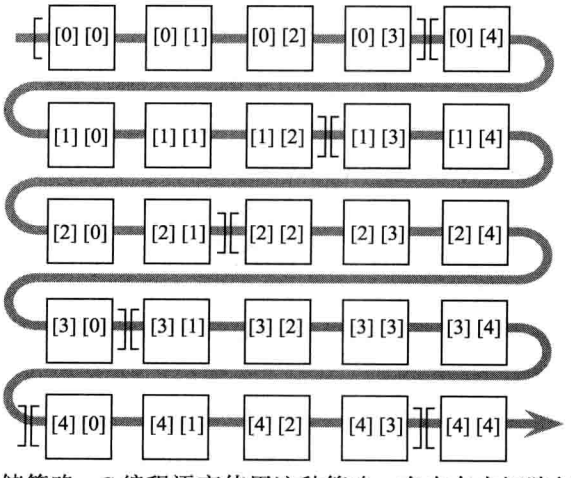


图 3-3 矩阵按行存储策略，C 编程语言使用这种策略。在内存中矩阵行被连续存储。假定 cache 行包含四个矩阵元素（使用中括号表示）

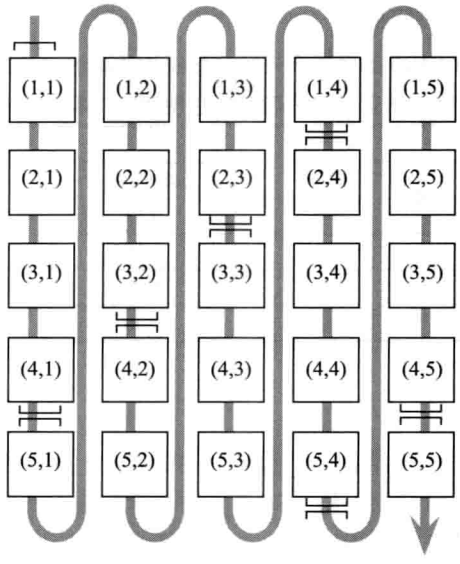


图 3-4 矩阵按列存储策略，Fortran 编程语言使用这种策略。在内存中矩阵列被连续存储。假定 cache 行包含四个矩阵元素（使用中括号表示）

70

3.3 案例分析：Jacobi 算法

Jacobi 算法是数值分析和模拟中许多基于 stencil 循环方法的原型。在其最简单的形式中，可以用来求解一个标量函数 $\Phi(\vec{r}, t)$ 的扩散方程：

$$\frac{\partial \Phi}{\partial t} = \Delta \Phi \tag{3-6}$$

求解一个长方形区域的狄利克雷边界条件。当使用有限差分法时，其微分算子是离散的（在不丧失一般性的前提下，这里我们限定为二维方程。但请分析习题 3.4 中，二维和三维性能的区别）。

$$\frac{\delta \Phi(x_i, y_i)}{\delta t} = \frac{\Phi(x_{i+1}, y_i) + \Phi(x_{i-1}, y_i) - 2\Phi(x_i, y_i)}{(\delta x)^2} + \frac{\Phi(x_i, y_{i-1}) + \Phi(x_i, y_{i+1}) - 2\Phi(x_i, y_i)}{(\delta y)^2} \tag{3-7}$$

在每一个时间步长，Φ 在坐标 (x_i, y_i) 处的修正值 $\delta\Phi$，由式（3-7）使用其四个邻居点的原值计算获得。当然，Φ 的新值必须写回第二个数组中。当所有点都完成更新后，算法进入下一层迭代。代码清单 3-1 是该内核的一个可能实现。该实现解决了稳定状态的问题但缺乏收敛条件，当然这不是我们考虑的重点（注意交换 $t0$ 和 $t1$ 区域不需要逐个元素交换。同该算法的初步实现相比，仅通过交换第三个数组的索引，我们就获得了两倍的性能提升）。

许多优化方法都可能加速这段代码。我们首先使用平衡分析预测这段代码的性能，然后和实测值相比较。图 3-5 说明了一个二维 Jacobi 算法的五点更新过程。每次更新需要四次数据加载和一次数据存储操作，但是其“下行邻居”（downstream）phi(i+1, k, $t0$) 在两次迭代之后一定会再次用到（从 cache 中），所以在计算代码平衡值时，只有三次数据读取操作（共有

四次）：B_c = 1.0 W/F（如包含写分配，则代码平衡值为 1.25 W/F）。然而，如图 3-5 所示，按行遍历会将 k 坐标最大（即 phi(i,k+1,t0)）的元素第一次加载到 cache 中。这是在内存传输中不可避免的。如果 cache 足以容纳超过两行的数据元素，那么在 cache 中三次连续的行遍历会使用这个元素。在这种情况下，我们可以假设加载 k 和 k−1 行没有时间消耗，所以代码平衡值降为 B_c = 0.5 W/F(如果包含写分配，其值为 0.75 W/F)。如果内层维度逐渐变大（如图 3-5 所示，列变多），必然要增加一个，最终三个额外的数据加载操作导致代码平衡值回到令人不愉快的 B_c = 1.0(1.25) W/F。 71

代码清单 3-1 二维 Jacobi 算法的简单实现

```
1  double precision, dimension(0:imax+1,0:kmax+1,0:1) :: phi
2  integer :: t0,t1
3  t0 = 0 ; t1 = 1
4  do it = 1,itmax     ! choose suitable number of sweeps
5    do k = 1,kmax
6      do i = 1,imax
7         ! four flops, one store, four loads
8         phi(i,k,t1) = (  phi(i+1,k,t0) + phi(i-1,k,t0)
9                        + phi(i,k+1,t0) + phi(i,k-1,t0) ) * 0.25
10     enddo
11   enddo
12   ! swap arrays
13   i = t0 ; t0=t1 ; t1=i
14 enddo
```

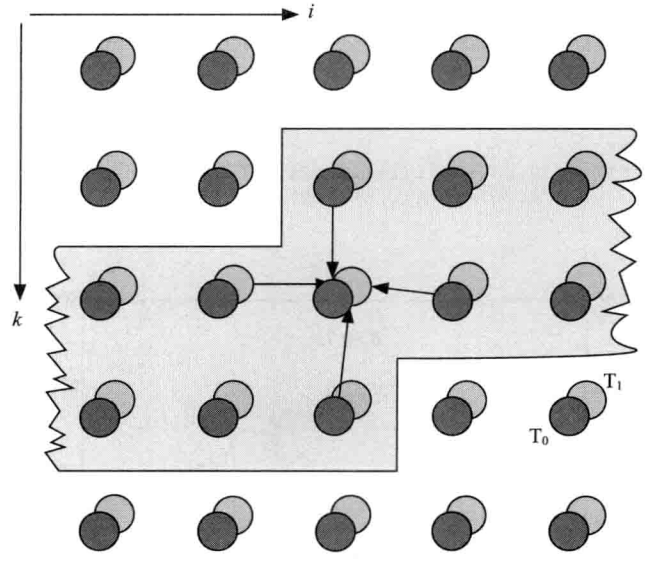


图 3-5 二维 Jacobi 算法更新示意图。如果至少有连续两行可以保存在 cache 中（阴影区域），在每次更新时，只有 T_0 点需要从内存中获得（交叉阴影点） 72

根据这些平衡因子，我们可以计算出给定硬件架构上代码的 lightspeed。3.1.2 节已经给出了在英特尔 Xeon 5160 平台上的 STREAM 结果。当 cache 足以容纳两个连续行时，数据传输特点和 STREAM COPY 以及 SCALE 相匹配：一次数据加载、一次数据存储，再加上一次强制性写分配。因为 Jacobi 内核只包含一次乘法操作，而不是三次加法操作，所以理论的机器平衡值 B_m = 0.111 W/F 不得不进行修改。因此我们使用

$$B_{\mathrm{m}}^{+}=\frac{0.111}{4/6}\ \mathrm{W/F}\approx 0.167\mathrm{W/F} \tag{3-8}$$

基于这个理论值，并假定写分配不能避免，因此：

$$l_{\mathrm{best}}=\frac{B_{\mathrm{m}}^{+}}{B_{\mathrm{c}}}=\frac{0.167}{0.75}\approx 0.222 \tag{3-9}$$

其中，修正的理论峰值性能为 P_{max}^{+} = 12 · 4/6GFlop/s = 8 GFlop/s，则预测性能为 1.78GFlop/s。然而，基于表 3-3 列出的 STREAM COPY 值，该预测性能应再乘以 0.38，实际预测性能为 675 MFlop/s。随着内层维度的增大，cache 已经不能够同时容纳两个甚至一个连续行，代码平衡值第一次上升为 B_{c} = 1.0 W/F，最终为 B_{c} = 1.25 W/F。图 3-6 显示了附加各种限制和预测的、不同内层维度的性能对比图（为节省内存，当 N 较大时（即 kmax<<imax）使用了非方形区域）。这个模型可以非常明显地描述整体行为。小规模的性能波动可以有多种原因，例如合并或者内存 bank 的影响。

图 3-6 同时也引入一个新的性能指标：更新的区域数目（LUP）/ 秒。因为它强调“工作完成”而不是 MFlop/s，所以更适合 stencil 算法。在我们的案例分析中，Flop 和 LUP 间有
73 一个简单的 1 : 4 的对应关系。但在一般情况下，MFlop/s 会随着算法优化方法的使用而变化，例如使用不同编译器重新排列代码序列而导致每次更新所要执行的浮点数运算数目不同。然而，用户最感兴趣的是在一定时间内可以完成多少实际工作。LUP/s 使得当底层问题相同时，不管采用何种优化方法，所有的性能都具有可比性。例如，许多处理器提供了 Fused Multiply-Add（FMA）机器指令执行 $r = a + b \cdot c$ 操作：一次可执行两次浮点数运算。在这种情况下，因为每次浮点数运算延迟的减少，FMA 可大幅提高性能。将代码清单 3-1 Jacobi 算法代码重写如下：

```
1 do k = 1,kmax
2   do i = 1,imax
3     phi(i,k,t1) =  0.25 * phi(i+1,k,t0) + 0.25 * phi(i-1,k,t0)
4                  + 0.25 * phi(i,k+1,t0) + 0.25 * phi(i,k-1,t0)
5   enddo
6 enddo
```

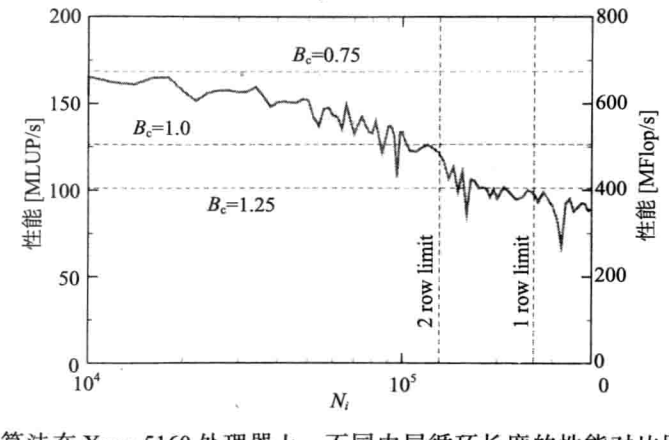


图 3-6 Jacobi 算法在 Xeon 5160 处理器上，不同内层循环长度的性能对比图。横线为基于 STREAM 基准测试带宽的性能预测

这个版本用 7 次浮点数运算代替了 4 次浮点数运算；当算法是访存受限时，MLUP/s 性能指标并没有发生变化（这留给读者使用平衡分析方法证明），但是 MFlop/s 却发生了变化。

3.4 案例分析：稠密矩阵转置

在下面的实例分析中，假定矩阵按列存储。计算一个稠密矩阵的转置（$\boldsymbol{A} = \boldsymbol{B}^{\mathrm{T}}$），根据循环的组织顺序，矩阵 $\boldsymbol{A}$ 或者 $\boldsymbol{B}$ 会有一个矩阵的访存是非连续的。矩阵转置最不幸的实现方式如下：

```
1 do i=1,N
2   do j=1,N
3     A(i,j) = B(j,i)
4   enddo
5 enddo
```

矩阵 $\boldsymbol{A}$ 的写入操作是非连续的（见图 3-7）。由于写分配操作的影响，非连续写比非连续读的代价要大得多。从这个最坏的代码出发，我们尝试获得期望性能。由于矩阵转置不执行任何算术操作，因此我们使用有效带宽（即应用程序达到的 GB/s）来表示性能。

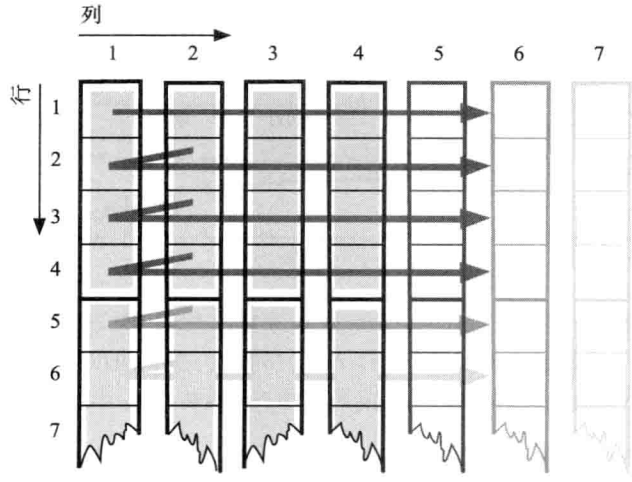


图 3-7 vanilla 矩阵转置中的 cache 行遍历（非连续写入操作，按列存储）。如果矩阵第一维的大小是 cache 行的整数倍，则矩阵的每一列都是 cache 行对齐的

C 表示 cache 大小，L_c 表示 cache 行大小（单位为 DP 字）。根据矩阵大小的不同，共有三个主要的性能优化法则：

- 当两个矩阵可以一次性加载到 CPU cache 中时（$2N^2 \lesssim C$），我们期望的有效访存带宽同 cache 的访存速度是同一数量级的。空间局部性的重要性只体现在多个不同的 cache 层次上。这种情况下，优化空间有限。 74
- 当矩阵太大而不能一次性加载到 cache 中，但仍然有

$$NL_c \lesssim C \tag{3-10}$$

$\boldsymbol{A}$ 的非连续写操作对性能的影响微乎其微，这是因为在一次完整的行遍历中，所有导致写未命中数据的写入操作都开启一个 cache 行进行写分配。这些 cache 行在接下来的 L_c-1 行中有很大可能依然位于 cache 中，减轻了非连续写的影响（空间局部性）。在这种情况下，有效访存带宽应该和处理器可达到的最大访存带宽位于同一数量级。

- 当 N 继续增大，使得 $NL_c \gtrsim C$ 时，每次对 $\boldsymbol{A}$ 的写入操作都会导致 cache 未命中和随后的写分配操作。在这种情况下，当进行内存写入时，只有一个 cache 行的数据被真

正用到，所有的空间局部性都会丧失，因此导致预测性能的大幅下跌。

图 3-8 的 vanilla 部分说明了上述假设基本上是正确的。然而，即使工作集全部加载到 cache 中，非连续写操作看起来也是非常低效的。这可能是因为所选择架构（英特尔 Xeon/Nocona）的 L1 cache 是完全写入（write-through）类型；例如，不管 L1 cache 命中与否，L2 cache 在写入操作时总是更新。因此，两级 cache 间的写分配操作浪费了大部分内部可用带宽。

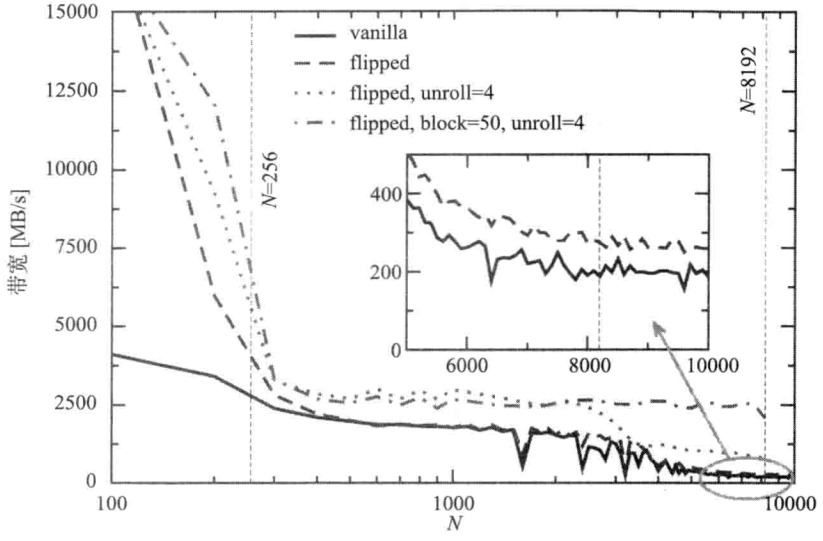


图 3-8 稠密矩阵转置的不同实现在 L2 cache 大小为 1MB 的现代处理器（英特尔 Xeon/Nocona 3.2 GHz）上的性能（有效带宽）图。$N = 256$ 代表了矩阵可以完全加载到 cache 中的临界大小，$N = 8192$ 代表了 N 个 cache 行可完全加载到 cache 中的临界大小

在上述的第二个性能优化法则中，当写入操作的 N 个 cache 行占用的 cache 大小和 L2 cache 大小具有可比性时，性能会大致恒定在一个水平上：有效带宽大约是 1.8 GB/s，同理论最大带宽 5.3 GB/s（由两个性能为 333MTransfer/s 的内存通道获得）相比并不理想。绝大多数商业架构的理论峰值带宽通过使用编译器生成的代码不可能达到，但是 50% 的性能
75 还是经常可以达到的。因此，肯定有一个因素能够显著降低可用带宽。这个因素就是快表（Translation Lookaside Buffer，TLB），物理地址和逻辑地址转换表。TLB 可以看作是额外的一级 cache，其 cache 行大小为物理内存页面大小（页面大小通常为 4KB，也有 16KB 的，在有的系统上甚至可以配置）。在本节使用的架构上，它只需容纳 64 个元素，对应于 256KB 内存（物理页面大小为 4KB）。这个值小于 L2 cache 大小，所以 TLB 的影响在 cache 内部也可以观察到。更为严重的是，当 N 大于 512 时，也即一个矩阵行大小超过了物理页面的大小，非连续访存的每一个单独访存操作都会导致 TLB 未命中。即使页表位于 L2 cache 中，这也会明显降低有效带宽。这是因为每一次 TLB 未命中都会导致至少 57 个处理器时钟周期的访存延迟（在这个给定的 CPU 上）。在频率为 3.2 GHz，总线传输速率为 666MW/s 的核上，这相当于传输超过 64 字节 cache 行的数据。

当 $N \gtrsim 8192$ 时，性能最终达到了预期的低水平。本节所使用处理器的理论内存带宽为 5.3GB/s，实际仅达到了大约 200MB/s。在有效长度为 128 个字节（每次未命中时，两个长度为 64 字节的 cache 行被一起读取，但分别提取）的 cache 行中，只有一个元素被用来执

行非连续写操作。在 cache 内部，每一次循环迭代都需要读取或者写入 3 个字，但最坏的情况是需要读取或者写入 33 个字。因此我们期望一个 1：11 的性能比，同观察到的值大略相等。 76

我们必须再次强调基于硬件特征的性能预测并不是在任何情况下都能够发挥作用 [M41, M44]，特别是在商业系统中，其芯片组、内存芯片和中断基本上不可控。有时对发生的独特性能行为的原因只有一个定性的了解，但这足以获得下一步的逻辑优化步骤。

稠密矩阵转置最重要的一个优化方法是交换嵌套循环的顺序，即将 i 移入内层循环。虽然这样会导致矩阵 ***B*** 的非连续读，但是却消除了矩阵 ***A*** 的非连续写，从而节省了大约一半（准确地说是 5/11）的内存带宽（对于较大的 N）。实际的性能收益（见图 3-8 “flipped” 部分）虽然明显，但还是没有达到这个性能预期。一个可能的原因是缺乏一个对于非连续写的更高效的内存接口。

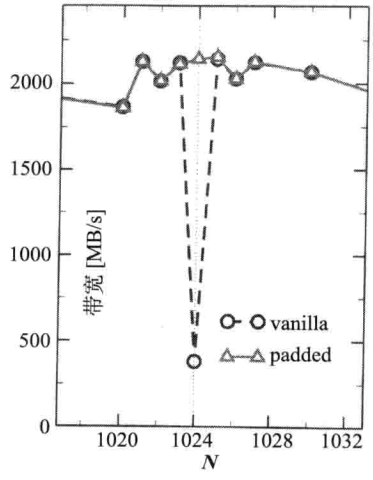


图 3-9 数组维度大小的不明智选择造成的 cache 颠簸（虚线）：矩阵转置性能在矩阵大小为 1024*1024 时显著降低。通过 padding 方法使第一维度大小增大 1 倍可完全消除颠簸（实线）

图 3-8 显示的性能图在某些点看起来非常不稳定。粗略看来，并不确定是否是某些 N 值导致了明显的性能降低（同它的邻值比较）。然而，通过仔细分析（图 3-9 的 “vanilla” 部分）可以看出当数组维度是 2 的指数倍时，对性能是非常不利的（基准测试程序对于每个新的 N 都会分配合适大小的矩阵）。像在 1.3.2 节描述的那样，由于关联性的不足，当连续迭代命中相同的 cache 行时，非连续访存会导致性能 “颠簸”。图 3-7 非常清楚地说明了当矩阵的第一维大小是 2 的指数倍时，转置操作是非常容易发生性能 “颠簸” 的。对于大小为 C 并且直接映射的 cache，每隔 C/N 次迭代就会命中相同的 cache 行。对于大小为 L_c 的 cache 行，有效的 cache 大小为： 77

$$C_{\text{eff}} = L_c \max\left(1, \frac{C}{N}\right) \tag{3-11}$$

由于关联性的限制，这是实际可用的 cache 大小。对于一个 m 路组关联的 cache，其有效大小也仅仅是乘以 m。考虑一个真实的例子：$C = 2^{17}$ (1 MB)，$L_c = 16$，$m = 8$，$N = 1024$，$C_{\text{eff}} = 2^{11}$ DPW（DP word，DP 字），即 16 KB，则 $NL_c >> C_{\text{eff}}$，其性能应该和 N 值很大时相似。

然而，一个简单的代码变换就可以消除性能 “颠簸” 的影响：假定矩阵的大小为 1024*1024，将第一维的大小增加 p（称为 padding），得到（1024+p，1024）的矩阵 ***A***，从而导致完全不同的 cache 使用模式。如果 $Cm/N > L_c/p$（见图 3-10），则 L_c/p 次迭代后，访问地址属于另外一组 m 个 cache 行，并且没有关联性冲突。当将第一维的大小增加 1 时，图 3-9 显示了显著的效果。通常来讲，应该使用所有的方法使数组的第一维大小不是 2 的指数倍。不同的维度添加不同大小的 padding 从而获得最优性能是非常有效的。所以有时可应用一个经验法则：尽量将数组的维度修正为 16 的奇数倍。

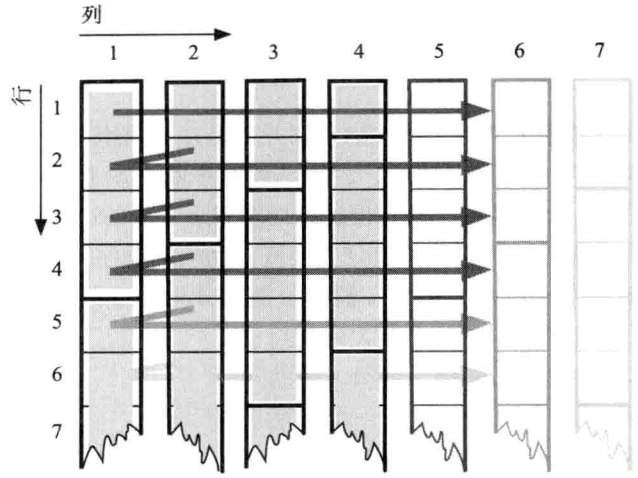


图 3-10 padding 矩阵转置的 cache 行遍历。通过减少关联冲突，padding 可能会增加有效 cache 大小

78 应用矩阵转置算法的更进一步优化方法将在下面的章节中详细讨论。

3.5 算法分类和访存优化

在基于 cache 的处理器上，许多循环的优化潜力可以通过观察某些基本参数（像数据传输、算术运算和问题规模的伸缩性）很容易地估算出来。从而进一步确定优化的努力是否有意义。

3.5.1 O(N)/O(N)

如果算术运算量和数据传输量（加载 / 写入）与问题规模（或者“循环长度”）N 成比例，那么优化潜力是非常有限的。标量乘、向量加和稀疏矩阵向量乘都是这类问题的典型实例。当 N 取值较大时，其性能不可避免地受访存限制，并且使用编译器生成的代码就可以达到较高的性能。这是因为 $O(N)/O(N)$ 循环往往很简单，正确的软件流水策略的作用非常明显。但是嵌套循环是一个不一样的问题（具体见下面的分析）。

然而，即使循环不嵌套，优化空间还是经常存在的。作为一个实例，考虑下面的向量加代码：

```
1 do i=1,N
2   A(i) = B(i) + C(i)
3
4 do i=1,N
5   Z(i) = B(i) + E(i)
6 enddo
```

循环整合 →

```
! optimized
do i=1,N
  A(i) = B(i) + C(i)
! save a load for B(i)
  Z(i) = B(i) + E(i)
enddo
```

左边代码的每一个循环都没有优化空间。每个循环包含两次数据读取操作、一次数据写入操作和一次加法操作（没有计算写分配），因此循环代码均衡值为 3/1。然而，数组 B 在第二个循环中被重新加载了一次，这是非常没有必要的：将两个循环整合为一个，数据 B 的元素就可以只被读取一次，循环代码平衡值降为 5/2。在其他条件都一样的情况下，代码性能（访存受限）将会提高 6/5（如果写分配时不可避免，这个值将是 8/7）。

对于这两个循环来说，循环整合实现了 $O(N)$ 的数据重用，减少了一个数组的数据读取

操作。像这种简单情况，编译器一般会自动使用该优化方法。

3.5.2 $O(N^2)/O(N^2)$

典型的循环长度为 N 的两层嵌套循环中，有 $O(N^2)$ 算术运算操作和 $O(N^2)$ 数据读取和写入操作。稠密矩阵向量乘、矩阵转置以及矩阵加都是这类问题的典型实例。尽管内层循环的情况和 $O(N)/O(N)$ 相似，并且都访存受限，但嵌套却开启了新的优化空间。然而，优化又一次仅限于一个常数因子的改善。考虑下面的稠密矩阵向量乘（Matrix-Vector Multiply, MVM）代码： 79

```
1 do i=1,N
2   tmp = C(i)
3   do j=1,N
4     tmp = tmp + A(j,i) * B(j)
5   enddo
6   C(i) = tmp
7 enddo
```

上面代码的代码平衡值为 1W/F（矩阵 ***A***、***B*** 两次数据读取和两次算术运算）。数组 C 由外层循环变量索引，所以其更新可以在寄存器中进行（这里我们通过使用标量 tmp 来说明，尽管编译器可以自动进行这些转换），从而不计算其读取和写入操作。矩阵 ***A*** 只被加载一次，但是 ***B*** 却被加载了 N 次（见图 3-11，外层循环每迭代一次，***B*** 就被加载一次）。我们可以采用上节所讲的循环整合方法，但是这里需要整合的内层循环有 N 个而不是两个。解决方案是循环展开：外层循环间隔 m 遍历，内层循环重复 m 次。我们必须要面对外层循环长度不能被 m 整除的情况。在这种情况下，要单独处理剩余的循环。

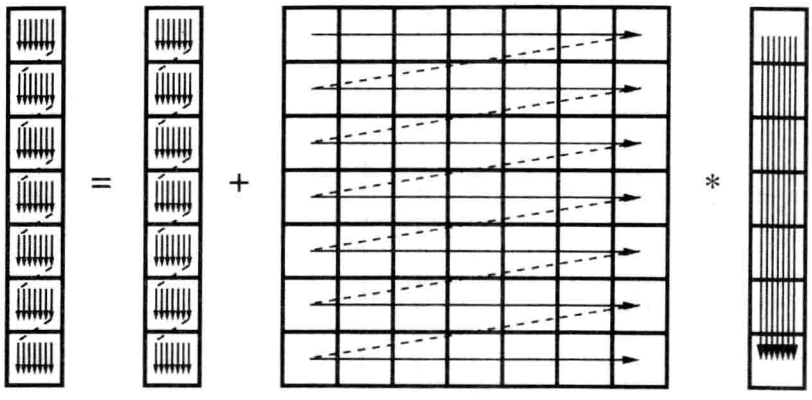


图 3-11 未优化的稠密矩阵向量乘（$N \times N$），RHS 向量被加载了 N 次

```
1  ! remainder loop
2  do r=1,mod(N,m)
3    do j=1,N
4      C(r) = C(r) + A(j,r) * B(j)
5    enddo
6  enddo
7  ! main loop
8  do i=r,N,m
9    do j=1,N
10     C(i) = C(i) + A(j,i) * B(j)
11   enddo
```

80

```
12    do j=1,N
13      C(i+1) = C(i+1) + A(j,i+1) * B(j)
14    enddo
15    ! m times
16    ...
17    do j=1,N
18      C(i+m-1) = C(i+m-1) + A(j,i+m-1) * B(j)
19    enddo
20  enddo
```

“剩余循环”可以采用同原始循环一样的优化方法，但这不重要了。我们在下面的讨论中将忽略这些“剩余循环”。

仅应用外层循环展开方法，除了代码膨胀外我们什么也没有得到。然而，现在可以非常容易地使用循环整合方法了：

```
1  ! remainder loop ignored
2  do i=1,N,m
3    do j=1,N
4      C(i) = C(i) + A(j,i) * B(j)
5      C(i+1) = C(i+1) + A(j,i+1) * B(j)
6      !  m times
7      ...
8      C(i+m-1) = C(i+m-1) + A(j,i+m-1) * B(j)
9    enddo
10 enddo
```

将外层循环展开，然后整合通常称为展开与合并（unroll and jam）。m 路展开与合并实现了寄存器中 $\boldsymbol{B}$ 每个元素的 m 次复用，代码平衡值减小为 $(m+1)/2m$。当 $m>1$ 时，该值显然小于 1。如果 m 取值非常大，由于 $\boldsymbol{B}$ 加载的次数大为减少，因此可获得接近两倍的性能提升。在理想情况下，$\boldsymbol{B}$ 只需加载一次。因为 $\boldsymbol{A}$（大小为 N^2）只需加载一次，m 路展开与合并的总数据传输量为 $N^2(1+1/m)+N$。图 3-12 显示了两路展开的稠密矩阵向量乘的访存模式。

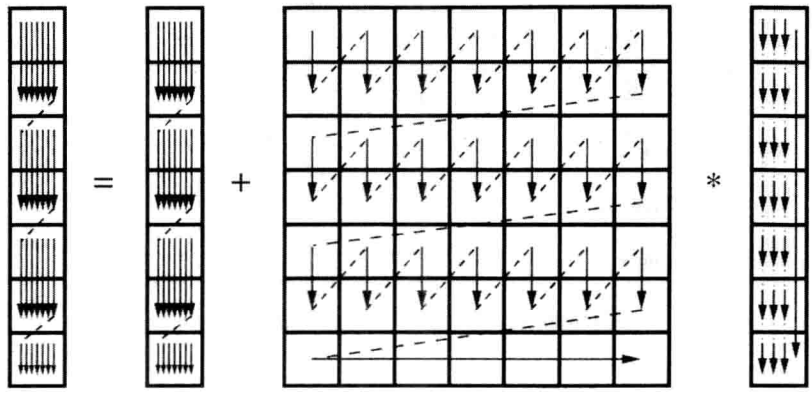


图 3-12 两路寄存器展开的稠密矩阵向量乘算法。重加载 RHS 向量造成的数据冲突被大约降低了一半，剩余的循环在外层调用本例

然而，所有这一切都假设寄存器的压力不会太大。即假设 CPU 有足够多的寄存器来容纳循环体内部数量相当可观的操作数。如果不能，编译器必须溢出寄存器数据到 cache 中，这样会降低计算性能（见 2.4.5 节）。再一次强调，可用的编译日志可以帮助确定这种情况。

循环展开与合并在高级别优化上可由有些编译器自动实现。一个复杂的循环体可能会隐藏一些重要的优化信息，因此手工优化是必需的：通过手工优化或者编译器指令来指定像循环展开等高级别转换。如果可用，编译器指令是首要选择的优化方法。这是因为指令比较容易维护且不会导致明显的代码膨胀。可惜的是，编译器指令本质上是不可移植的。

尽管同稠密矩阵向量乘相比，上节所讨论的矩阵转置算法没有直接机会优化数据传输（两个矩阵只需读写各一次），该算法同样是 $O(N^2)/O(N^2)$ 问题的典型实例。通过在矩阵转置算法的“flipped”版本上应用循环展开与合并方法，得到了将近 50% 的性能提升（参见图 3-8 的虚线）。 81

```
do j=1,N,m
  do i=1,N
    A(i,j)     = B(j,i)
    A(i,j+1)   = B(j+1,i)
    ...
    A(i,j+m-1) = B(j+m-1,i)
  enddo
enddo
```

不要期望 $m = 4$ 时会有明显效果，因为基本性能分析没有发生变化：当 N 取值中等时，可用的 cache 行数目足够可以容纳 L_c 列数据。图 3-13 显示了 $m = 2$ 的情况。每一次加载操作中，cache 行中的 m 个元素可以连续访问。因此，尽管 TLB 依然太小而不能映射到整个工作集，但还是减少了 TLB 未命中。

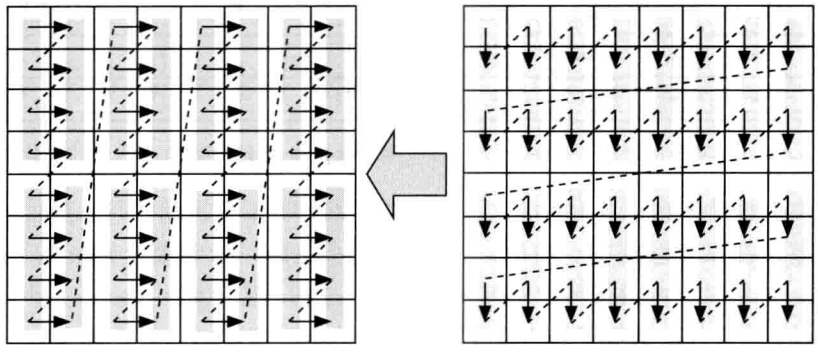


图 3-13 矩阵转置“flipped”版本的两路循环展开（例如，原始版本的跨距加载）

尽管如此，当 N 取值较大从而使 cache 不能容纳 N 个 cache 行的数据时，减少 TLB 未命中并不能阻止性能的下降。最好能有一个策略重用非连续 cache 行剩下的 L_c-m 个元素，从而使 cache 行可以很快替换出去而不需要再次加载。一个“野蛮”方法是 L_c 路循环展开，但是这个方法会导致写操作时更大间隔的非连续访问。同时，由于循环展开因子过大，导致循环体中寄存器使用量的增加（算术操作引起的），因此这并不是一个通用的优化方法。循环分块（loop blocking）能够在不增加寄存器使用的情况下实现 cache 的最优使用。该方法不会减少数据读取或者写入操作，但是会增加 cache 的命中率。对于深度为 d 的嵌套循环，分块引入了 d 个额外的外层循环，将原来的内层循环切分成块： 82

```
do jj=1,N,b
  jstart=jj; jend=jj+b-1
  do ii=1,N,b
    istart=ii; iend=ii+b-1
    do j=jstart,jend,m
      do i=istart,iend
```

```
7           a(i,j) = b(j,i)
8           a(i,j+1) = b(j+1,i)
9           ...
10          a(i,j+m-1) = b(j+m-1,i)
11        enddo
12      enddo
13    enddo
14  enddo
```

在这个例子中，除 m 路的循环展开与合并外，使用了二维分块（分块因子为 b）方法。这个变化并没有改变主循环体，因此并没有改变保存操作数的寄存器数目。然而，却极大改进了 cache 行的访存模式，如图 3-14 所示，两路循环展开与 4*4 分块的结合。如果分块因子选择恰当，那么非连续写操作中的 cache 行将会在每一个分块的最后得到充分利用，而且会快速地置换出去。因此，N 取值较大时的性能下降现象将不复存在。图 3-8 的虚线说明了 50*50 的分块结合 4 路循环展开消除了非连续内存访问带来的所有访存问题。

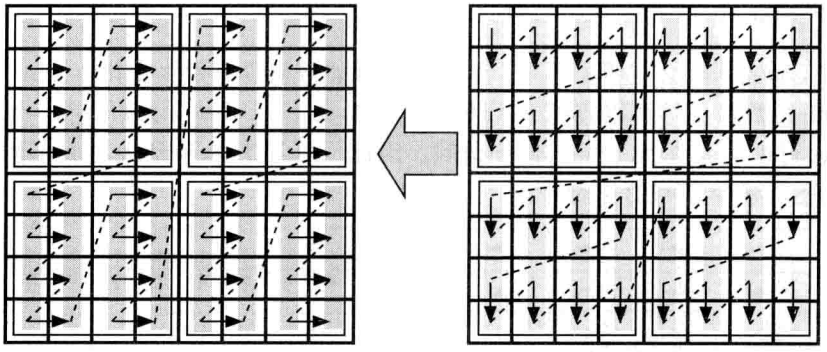


图 3-14 矩阵转置算法“flipped”版本应用 4*4 的分块和两路循环展开的优化方法

循环分块是非常普遍和强大的优化方法，并且编译器不会自动执行。最佳分块因子应该通过精心的基准测试实验获得。然而，cache 的大小也可以指导最佳分块因子的选择。即，
83 当对 L1 cache 分块时，内层循环中所有分块的大小总和不应该超过 cache 大小的一半。究竟对哪一级 cache 进行分块依赖于具体的算术操作，这里没有通用的指导建议。

3.5.3 $O(N^3)/O(N^2)$

如果算术计算量随着问题规模的扩大以一定的因子增长而大于访存数据量，那么这类算法具有巨大的优化空间。通过上述优化技术（循环展开与合并、循环分块）的使用，这类算法性能可能不会受 cache 限制。这类算法展现了 $O(N^3)/O(N^2)$ 特征，稠密矩阵矩阵乘（Matrix-Matrix Multiplication，MMM）和稠密矩阵对角化是这类算法的典型实例。开发一个精心优化的 MMM 程序已经超过了本书的范围，更不用说特征值计算。但是我们可以通过一个简单的实例（实际 $O(N^2)/O(N)$ 类型）来说明这类算法优化的基本原理。

```
1 do i=1,N
2   do j=1,N
3     sum = sum + foo(A(i),B(j))
4   enddo
5 enddo
```

上面代码的数据集大小为 $O(N)$，但是却进行了 $O(N^2)$ 运算（额外调用了 foo() 函数）。

其中 ***B*** 从内存中被加载了 N 次，所以总存数据传输量为 $N(N+1)$。使用 m 路循环展开与合并优化方法后，可将其减少到 $N(N/m+1)$。但是循环展开因子过大的缺点在上节中已经指出。而内层循环的分块（分块大小为 b）有两个作用：

```
1 do jj=1,N,b
2   jstart=jj; jend=jj+b-1
3   do i=1,N
4     do j=jstart,jend
5       sum = sum + foo(A(i),B(j))
6     enddo
7   enddo
8 enddo
```

- ***B*** 从内存中仅加载一次，可使 b 足够小以使 b 个元素可以加载到 cache 中，直到不需要它们为止。
- ***A*** 加载 N/b 次，而不是一次。

尽管 ***A*** 通过 cache 被加载了 N/b 次，但当前 ***B*** 分块被置换出 cache 的可能性非常低。这是因为根据 LRU 置换算法，当该 cache 行被频繁使用时是不会被置换出去的。这使得内存传输非常高效：共传输 $N(N/b+1)$ 个字。因为 b 的取值可以比典型循环展开因子大许多，所以分块是最好的优化策略。此外，循环展开与合并方法依旧可以用来增加 cache 内部（in-cache）的代码平衡值。基本的 N^2 依赖还是存在的，但是结合前因子可以说明内存受限和 cache 受限程序的差异。当内存带宽和访存延迟不是性能限制因素时，该代码是 cache 受限的。cache 受限代码在特定架构上能否达到预期性能目标依赖于 cache 大小、cache 和内存间的数据传输速度，当然还有算法本身。

$O(N^3)/O(N^2)$ 算法是经过精心优化、其性能可以接近硬件理论峰值性能的典型候选算法。如果循环分块和展开因子选择恰当，则对于 $N \times N$ 的矩阵（当 N 取值不会过小时），稠密矩阵向量乘的性能经常会达到峰值性能的 90%。系统厂商会提供该算法的高度优化版本，一般包含在 BLAS（Basic Linear Algebra Subsystem，基本线性代数子系统）库中。既然循环分块已经完成了将代码转化为 cache 受限的绝大部分重要工作，为什么还要应用循环展开方法？这是因为即使所有的数据都位于 cache 中，但许多处理器架构在每个时钟周期内也不能持续维持足够的加载和写入操作来满足算术运算的需要。例如，当前英特尔的 x86 架构处理器每个时钟周期内只能进行一次数据加载和一次写入操作，这就使得当内核的循环嵌套使用多于一个数据加载操作时（特别是上例中 $O(N^2)/O(N)$ 算法的 cache 受限分块），就需要使用循环展开与合并方法。

这里的讨论是出于教育目的，因此没有必要手工编写和优化标准线性代数和矩阵操作。这些函数一般总是从高度优化的函数库中调用。尽管如此，这里讨论的优化方法是能够应用到许多实际代码中去的。稀疏矩阵向量乘就是一个有趣的例子（见 3.6 节）。

3.6 案例分析：稀疏矩阵向量乘

上节讨论的稀疏矩阵向量乘是采用循环分块和展开优化方法的一个实际应用。该算法是大多数迭代矩阵对角化算法（Lanczos、Davidson、Jacobi-Davidson）的重要部分，而且经常会成为性能限制因素。当矩阵的非零元素数量 N_{nz} 随矩阵行数 N_r 的增加而线性增长时，这个矩阵称为稀疏矩阵。出于效率方面的考虑，内存中只存储稀疏矩阵的非零元素。因此，稀疏

MVM（sMVM）是一个 $O(N_r)/O(N_r)$ 问题，并且当 N_r 取值非常大时，该算法是访存受限的。尽管如此，循环嵌套使算法具有显著的优化潜力。图 3-15 所示的 sMVM 算法中，尽管矩阵的访存模式是非常有利的，但是 RHS 向量经常是非连续访问甚至是间接寻址的。下面章节将会进行不失一般性的讨论。

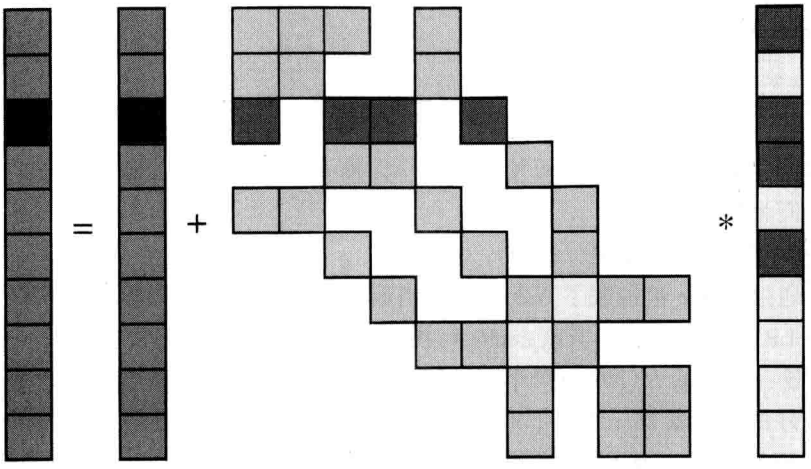


图 3-15 稀疏矩阵向量乘。图中黑色元素值只更新一个 LHS 元素。除非稀疏矩阵每行的第一个和最后一个非零元素间没有间隔，否则 RHS 向量的间接寻址是不可避免的

3.6.1 稀疏矩阵的存储机制

目前有多种稀疏矩阵存储机制，其中一些机制只适合存储特定类型的矩阵 [N49]。内存的访问模式和 sMVM 的性能特征严重依赖于所使用的存储机制。CRS（Compressed Row
86 Storage，压缩行存储）和 JDS（Jagged Diagonals Storage，锯齿对角线存储）是目前两个最重要也是最常用的存储机制。下面的讨论中，我们将会看到：CRS 适合基于 cache 的微处理器架构，而 JDS 则支持依赖和循环结构，该结构非常适用于向量化系统。

CRS 存储机制使用长度为 N_{nz} 的数组 val 存储矩阵所有的非零元素（按行存储，且元素间没有间隔）。所以必须提供两个额外的整型数组来指定数组 val 元素在原矩阵中所属的行、列：长度为 N_{nz} 的数组 col_idx 和长度为 N_r 的数组 row_ptr。前者存储了每个非零元素所属的列，后者存储了每行开始的索引（使用非零元素在 val 数组的下标）（参见图 3-16）。使用该存储机制完成 MVM 的初始代码非常简单：

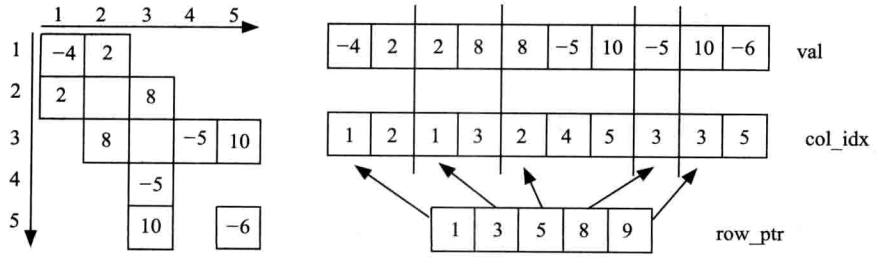


图 3-16 CRS 稀疏矩阵存储格式

```
1 do i = 1,Nr
2   do j = row_ptr(i), row_ptr(i+1) - 1
3     C(i) = C(i) + val(j) * B(col_idx(j))
4   enddo
5 enddo
```

下面几点需要引起注意：

- 外层循环迭代次数非常多（迭代次数为 N_r）。
- 相对于一般微处理器架构的流水线长度，内存循环迭代次数可能会比较少。
- 向量 ***C*** 的访存已经高度优化了：只从内存中加载一次。
- 数组 val 中非零元素的访存是连续的。
- 同期望的一样，RHS 向量 ***B*** 的访存模式为间接地址映射。然而，依赖于矩阵结构，这可能不是一个严重的性能问题。如果非零元素主要集中在对角线周围，则甚至会有明显的空间和时间局部性。
- B_c = 5/4 W/F（如果 col_idx 是 4 字节整型）。因为 cache 行的部分使用忽略了可能的大规模的数据传输。 87

其中有些点当我们后面演示并行 SMVM 时（参见 7.3 节）将很重要。

JDS 不仅要完成矩阵元素的消零，还需对矩阵元素进行重新组织。首先，移除矩阵中所有的零元素并将非零元素移到左边；其次将矩阵行按照非零元素的数目排序，使非零元素最多的行位于矩阵顶端，非零元素最少的行位于矩阵底部。矩阵行排序操作同时也生成置换映射数组并存储在长度为 N_r 数组 perm 中；最后，将矩阵的非零元素按列存储在数组 val 中。这些列也称为锯齿对角线（jagged diagonal，从原始稀疏矩阵的左上部遍历到右下部，参见图 3-17）。每一个非零元素在原矩阵的列索引像 CRS 一样存储在数组 col_idx 中。为使 RHS 和 LHS 向量元素的顺序相同，col_idx 数组根据置换映射数组重新生成。数组 jd_ptr 保存了

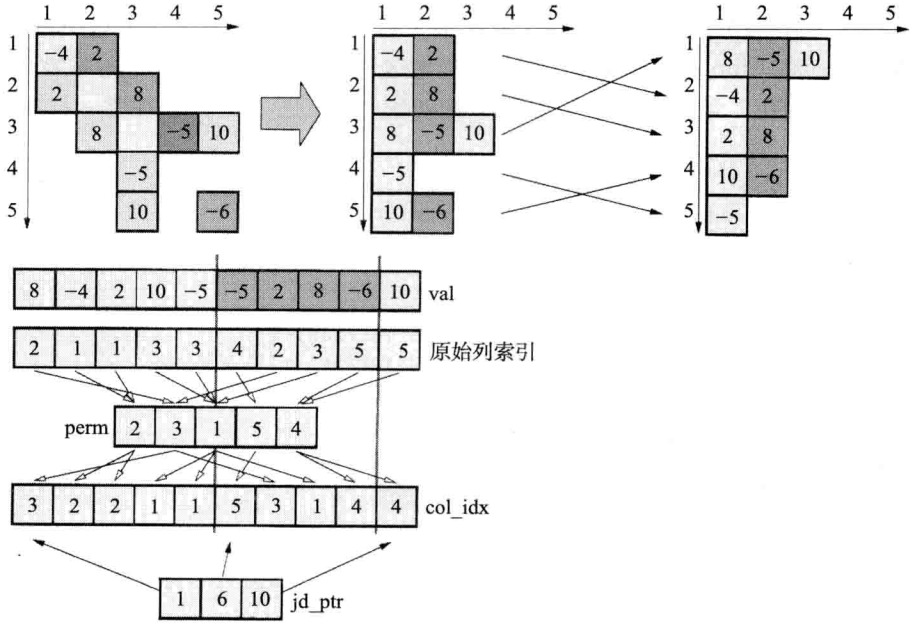


图 3-17 JDS 稀疏矩阵存储机制。列索引数组应用了置换映射（permutation map）技术。图中标记了其中一个锯齿对角线

第 N_j 条锯齿对角线的起始索引。采用 JDS 存储机制的 sMVM 的基本实现代码只比采用 CRS 稍微复杂一点：

```
1 do diag=1, Nj
2   diagLen = jd_ptr(diag+1) - jd_ptr(diag)
3   offset = jd_ptr(diag) - 1
4   do i=1, diagLen
5     C(i) = C(i) + val(offset+i) * B(col_idx(offset+i))
6   enddo
7 enddo
```

88

数组 perm 在这里没有被用到；通常情况下，所有 sMVM 操作都需要在置换空间内完成。这个循环有以下值得注意的属性：

- 内层循环的迭代次数非常多，且互相没有依赖。对于向量处理器来说，JDS 是比 CRS 更好的数据存储机制。
- 外层循环迭代次数较少（锯齿对角线的数目）。
- 结果向量从内存中被加载多次（至少是部分向量），所以这里可能有比较大的优化空间。
- 数组 val 中的非零元素是连续访存的。
- 同 CRS 一样，RHS 向量访存为间接地址映射。虽然一个良好的矩阵布局是使用直对角线而不是压缩行，但上述的注释依然可用。作为额外的复杂度，矩阵行和 RHS 向量都被置换了。
- B_c = 9/4 W/F（如果 col_idx 是 4 字节整型）。

从代码平衡值来看，sMVM 似乎更倾向于 CRS。

3.6.2 JDS sMVM 优化

JDS sMVM 要用到循环展开与合并技术，但是这个技术需要内层循环的长度独立于外层循环索引。不幸的是，锯齿对角线一般不会具有相同的长度，违反了这个条件。然而，一个称为循环剥离（loop peeling）的技术可解决这个问题：对于 m 路的循环展开，分割成 $m \times x$ 个块，剩下的少于 $m-1$ 个对角线可单独处理（参见图 3-18，像往常一样剩余循环被忽略）。

```
1  do diag=1,Nj,2  ! two-way unroll & jam
2    diagLen = min( (jd_ptr(diag+1)-jd_ptr(diag)) ,\
3                   (jd_ptr(diag+2)-jd_ptr(diag+1)) )
4    offset1 = jd_ptr(diag) - 1
5    offset2 = jd_ptr(diag+1) - 1
6    do i=1, diagLen
7      C(i) = C(i)+val(offset1+i)*B(col_idx(offset1+i))
8      C(i) = C(i)+val(offset2+i)*B(col_idx(offset2+i))
9    enddo
10   ! peeled-off iterations
11   offset1 = jd_ptr(diag)
12   do i=(diagLen+1),(jd_ptr(diag+1)-jd_ptr(diag))
13     c(i) = c(i)+val(offset1+i)*b(col_idx(offset1+i))
14   enddo
15 enddo
```

假定被剥离迭代的时间消耗可以忽略不计，则 m 路循环展开与合并将代码平衡值降低为：

89

$$B_c = \left(\frac{1}{m} + \frac{5}{4} \right) \text{W/F}$$

如果 m 取值足够大，则 JDS 代码平衡值就会接近于 CRS。然而，如之前讨论的那样，

较大的 m 值会导致寄存器使用量的增多，因此这一做法并不总是可取。通常情况下，一个合理的循环展开结合分块方法可以用来减少内存数据传输，同时提高 cache 内部性能。循环分块方法对于 JDS sMVM 来说也适用（见图 3-19）：

```
1  ! loop over blocks
2  do ib=1, Nr, b
3    block_start = ib
4    block_end   = min(ib+b-1, Nr)
5    ! loop over diagonals in one block
6    do diag=1, Nj
7      diagLen = jd_ptr(diag+1)-jd_ptr(diag)
8      offset = jd_ptr(diag) - 1
9      if(diagLen .ge. block_start) then
10       ! standard JDS sMVM kernel
11       do i=block_start, min(block_end,diagLen)
12         B(i) = B(i)+val(offset+i)*B(col_idx(offset+i))
13       enddo
14     endif
15   enddo
16 enddo
```

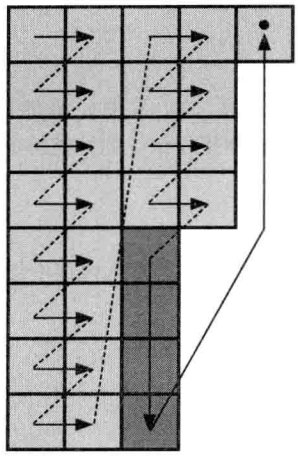


图 3-18 使用两路循环展开与合并和循环剥离方法的 JDS 矩阵遍历。被剥离的迭代被标记出来

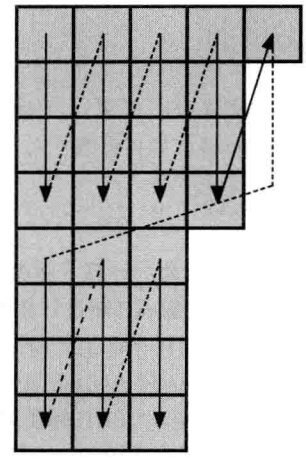


图 3-19 使用 4 路循环分块的 JDS 矩阵遍历

应用这个优化方法，当分块大小 b 取值不太大时，结果向量只需从内存中加载一次。尽管代码平衡值没有发生变化，但此时应该能够获得同 CRS 相近的性能。同之前论述的稠密矩阵转置一样，矩阵分块并没有优化寄存器的重用而是优化了 cache 的利用率。 90

图 3-20 显示了 CRS 与 JDS sMVM（原始、两路循环展开、大小为 400 的分块）在三个不同硬件平台上的性能对比。测试矩阵来源于固态物理学（半填充情形下六角一维 Holstein-Hubbard 模型）。CRS 似乎更适用于标准 AMD 和英特尔微处理器。这并不奇怪，因为它不需要手工优化就具有最低的代码平衡值，并且迭代次数较少的内层循环比较适合具有乱序执行能力的 CPU。然而，采用 EPIC 架构的英特尔 Itanium2 处理器 [V113] 对于 CRS 来说性能表现平平，但在分块的 JDS 版本中，其性能最高。因为缺乏乱序处理能力，编译器虽然检测到内层循环所有的指令集并行，但不能重叠一行的 wind-down 阶段和另一行的 wind-up 阶段，所以这个架构不能很好应付 CRS 中的短循环。当工作集可以加载到 cache 中时，这个效果会更加明显 [O56]。

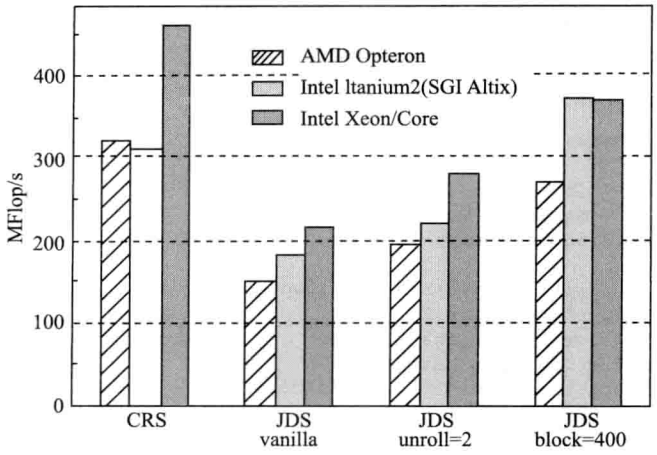


图 3-20 采用不用优化方法的 sMVM 性能对比图。选取了包含 1.7×10^7 个元素和 20 条锯齿的矩阵。对于不同的硬件架构来说，分块大小为 400 被证明是最优的

习题

3.1 *非连续访存*。如果一个或者多个数组以不定间隔方式读写，3.1 节中介绍的代码平衡值和 lightspeed 应如何修改？对于一个间隔为 s 的向量操作，可以期望什么样的性能特征？ 91

```
1 do i=1,N,s
2   A(i) = B(i) + C(i) * D(i)
3 enddo
```

3.2 *平衡值的乐趣*。计算下面循环内核的代码平衡值，假定所有数组都需要从内存中加载，并且忽略访存延迟的影响（超过计数变量 i 和 j 的循环行为是默认的）。

(a) Y(j) = Y(j) + A(i,j) * B(i)（矩阵向量乘）

(b) s = s + A(i) * A(i)（向量范数）

(c) s = s + A(i) * B(i)（标量乘）

(d) s = s + A(i) * B(K(i))（带间接访问的标量乘）

除数组 K 存储 4 字节的整型数外，所有数组都是双精度浮点类型。s 是一个双精度标量。根据理论峰值带宽和 STREAM 实测带宽（MFlop/s），计算这些内核在 Xeon 5160 处理器单核和 1.6 节描述的原型向量处理器上的期望性能。Xeon CPU 的 cache 行长度为 64 个字节。可以假设 N 足够大从而使数组不能全部加在到 cache 中。对于（d），请给出在 Xeon 处理器上最好和最坏的情形。

3.3 *性能预测*。未来主流微处理器架构的 SIMD 能力将会得到极大增强。其中一个可能的特征是 x86 处理器将能够在长度为 256 位（而不是 128 位）的寄存器上执行乘法和加法指令，也就是说可同时执行 4 个双精度浮点数的运算。这将会有效提高峰性能至两倍，如果 L1 cache 带宽也提高两倍，那么每个时钟周期执行的操作数将从 4 次提高到 8 次。假定其他参数如内存带宽和时钟性能保持不变，那么与当前英特尔“Core i7”（有效基于 STREAM 的机器平衡值为 0.12W/F）单核性能相比，评估可以得到的性能提升。假定一个完美的 SIMD 向量应用程序，其 60% 的计算时间代码平衡值为 0.04 W/F，40% 的计算时间代码平衡值为 0.5 W/F。如果厂商选择大力提升 CPU 的 SIMD 能力，例如，引进更长长度的向量。在这种情况下，什么会成为限制性能的绝对因素。

92 3.4 *优化三维 Jacobi 算法*。概括 3.3 节介绍的二维 Jacobi 算法，并考虑三维算法。变换内存循环的长度，你会期望性能特征的哪些改变（图 3-6）？参考 3.4 节介绍的稠密矩阵转置算法的优化，你能否得出消除性能下降的方法？

3.5 重新审视内存循环展开。到目前为止，我们遇到内存循环展开的可能性仅存在于软件流水和SIMD优化中（参见第2章）。内存循环展开在很多情况下是否也能够改进代码平衡值？通过内层循环展开提升Jacobi算子性能需要考虑哪些方面？

3.6 不能循环展开？考虑下面的下三角矩阵向量乘代码：

```
1 do r=1,N
2   do c=1,r
3     y(r) = y(r) + a(c,r) * x(c)
4   enddo
5 enddo
```

能否用展开并合并技术作用于外层循环（参见3.5.2节）来减少代码平衡值？尝试编写上面代码的四路展开版本。N没有特定的假设（除了N取值为正），矩阵A下三角（包括对角线）之外的所有元素都不能访问。

3.7 应用程序优化。对于下面的代码，你建议用什么优化策略？尝试修改下面代码，使其能够达到最高性能。

```
1 double precision, dimension(N,N) :: mat,s
2 double precision :: val
3 integer :: i,j
4 integer, dimension(N) :: v
5 ! ... v and s may be assumed to hold valid data
6 do i=1,N
7   do j=1,N
8     val = DBLE(MOD(v(i),256))
9     mat(i,j) = s(i,j)*(SIN(val)*SIN(val)-COS(val)*COS(val))
10  enddo
11 enddo
```

对于N没有任何假设。然而，你可以假设这是一段会被频繁调用的子程序，s和v在不同的调用中可能会发生变化，且v的所有元素都为正值。

3.8 TLB的影响。即使最现代的处理器，也没有足够大的快表可以存储驻留在外层cache上所有内存页面的映射。为什么TLB会如此小？这难道不是一个设计中的性能瓶颈吗？使用大页有什么好处？

93
~
94

第 4 章

Introduction to High Performance Computing for Scientists and Engineers

并行计算机

所谓的并行计算即许多“计算单元”（或核，core）协调解决同一个问题。所有现代超级计算机体系结构非常依赖并行结构，并且在 CPU 的数量越来越多。Top 500 通常用来评价超级计算机的速度 [W121]，它是基于 LINPACK 基准测试集上的性能数据为并行计算机排序，每年发布两次。LINPACK 可以解决未指定大小的稠密线性方程组系统，由于其只针对于单体系结构峰值性能方面的测试，因而不是一个被广泛接受的指标。也有一些其他的替代品，如 HPC 挑战基准测试集 [W122]，但是 LINPACK 提供了高效的开源代码，简单易用，所以近 20 年一直作为 Top 500 排序主要使用的基准测试集。Top 500 仍然是超级计算发展趋势的一个重要指标。从 Top 500 中超级计算机系统的处理器数量（见图 4-1）可以清晰地看到超级计算机的趋势：顶级线性 HPC 系统的性能不再只是依赖摩尔定律，而并行性变得越来越重要。近年来由于多核处理器的问世加速了这一趋势，最新的 Top 500 中已经不再有单核系统（可以参考 1.4 节）。当然，我们提供的是目前并行计算机技术的不完全介绍，可以参考由 Van der Steen 和 Dongarra 定期更新的“Overview of recent supercomputers”[W123]。

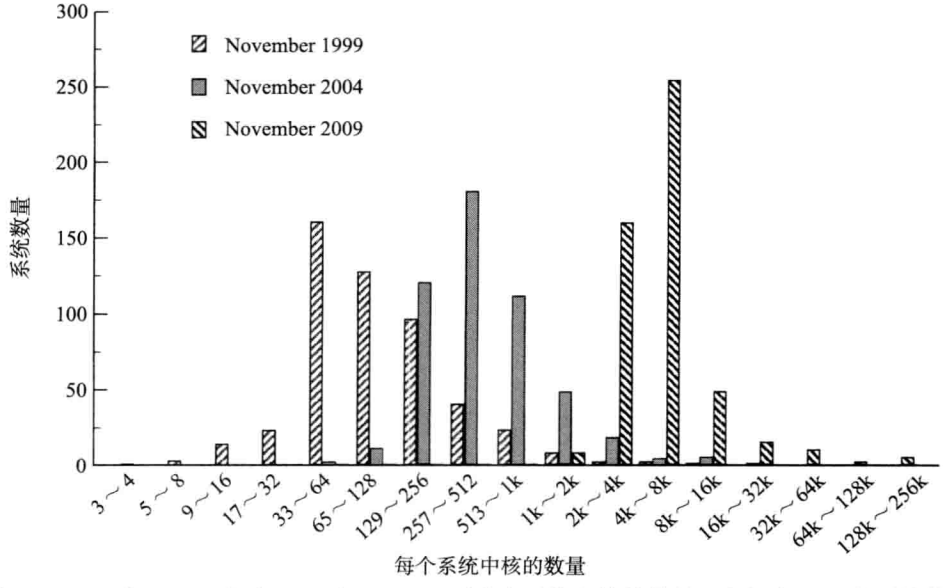


图 4-1　1999 年、2004 年和 2009 年 Top 500 列表中系统和核的数量。十年中 CPU 的平均数量增加了 50 倍。2004 ～ 2009 年，多核芯片的出现导致了核数量的戏剧性增长。数据来自 [W 121]

本章将介绍两种基本的并行计算机：共享内存和分布式存储类型。两种都要使用处理器或者更一般的说是“计算元素”之间的通信网络。因此我们也将概述基本通信网络的设计规则和性能特点。

4.1 并行计算模式分类

在并行架构中，一种广泛使用的分类标准是 Flynn 提出的根据并发控制和数据流的数量关系来分类。主要的概念是 SIMD 和 MIMD：

SIMD 单指令多数据。在单核处理器或者多核处理器上，一条单指令流同时并行处理多个数据流。例如向量处理器（见 1.6 节）、具有 SIMD 功能的现代超标量微处理器（见 2.3.3 节）、图形处理器（GPU）等。从历史上看，已经几乎绝迹的 SIMD 并行的大型多处理器都是以思维机（thinking machine）的连接机（connection machine）形式实现。

MIMD 多指令多数据。在多核处理器上，多条指令同时操作不同的数据。本章的共享内存和分布式存储并行计算机是典型的 MIMD 模式。

实际上还有另外两种分类：SISD（单指令单数据）和 MISD（多指令单数据）。前者一般描述在传统的存储程序数字计算机上非并行、单处理器执行方式，而后者在实践中并不是一个有效的模式。

严格来说，在超标量处理器中管道执行的指令级并行处理器（见 1.2.3 节及 1.2.4 节）并不在我们的分类中，虽然有些人可能说它应该属于 MIMD。然而，接下来我们严格地限制微处理器 MIMD 并行内置为共享或分布式存储并行计算机。 96

4.2 共享存储计算机

若干 CPU 工作在一个共享的物理存储空间的计算机就是共享存储计算机（shared-memory parallel computer）。虽然在功能上对程序员透明，但是在主存访问方面，共享存储系统带来了两种很不同的特点：

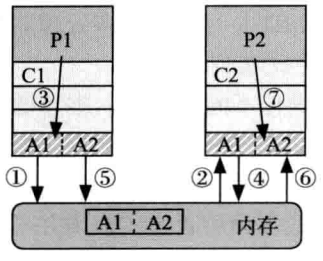
- 一致内存访问（Uniform Memory Access，UMA）模式是一个“flat”存储模式：所有的处理器和存储单元有着相同的延迟和带宽，这也称作对称多处理器（Symmetric MultiProcess，SMP）。在本书写作时，单个多核处理器芯片（见 1.4 节）只有“UMA 机器”，但是已经出现同一芯片不同的核组之间分别配置存储控制器的“片上集群”设计。
- cache 一致的非一致内存访问（ccNUMA）模式：存储在物理上是分布式的，但在逻辑上共享。这样的系统在物理分布上与分布式存储类似（见 4.3 节），但由网络逻辑把所有存储凝聚在一个单独的地址空间。由于分布式的特点，CPU 访问的位置不同（本地或者远程）会带来访存性能的不同。

由于存在多 CPU，不同 cache 中可能存储着同一个 cache 行的备份，其中有些备份很可能被修改。所以以上两种模式的 cache 一致性协议必须保证 cache 中的数据与内存中的数据在任何时候都相同。关于 UMA、ccNUMA 和 cache 一致性机制在后续章节中会详细介绍。在科学计算中共享存储的编程模式现在主要是 openMP，将在第 6 章介绍。

4.2.1 cache 一致性

不论是 UMA 模式还是 ccNUMA 模式，在所有存在 cache 的多核系统中 cache 一致性机制都是必需的。这是因为相同 cache 行的备份有可能同时存在于不同 CPU 的 cache 中，如果其中一个备份被修改并写到了内存中，则其他 cache 中的备份就过时了。cache 一致性协议要确保在所有情况下数据都相同。

图 4-2 给出了一个这样的例子。有两个处理器 P1、P2，分别有 cache C1 和 C2。每个 cache 行能存储两个元素。A1 和 A2 在内存中属于同一个 cache 行，分别被 P1 和 P2 修改。假设没有 cache 的一致性，每一个 cache 从内存中读这个 cache 行，然后在 C1 中 A1 被改
97 变，在 C2 中 A2 被改变，一段时间之后修改后的 cache 行将会写进内存。由于内存流的处理是以 cache 行大小为单位进行的，所以没有办法确定内存中正确的 A1 和 A2 值。



① C1 需要 cache 行的专有所有权
② 设 C2 中的 cache 行为 I 状态
③ C1 中的 cache 行已经为 E 状态→改变 C1 中的 A1 并设为 M 状态
④ C2 需要 cache 行的专有所有权
④ 写 C1 中的 cache 行然后将状态设为 I
⑥ 把该 cache 行加载进 C2，设状态为 E
⑦ 修改 C2 中的 A2，设 C2 中的状态为 M

图 4-2 两个处理器 P1、P2 分别在其 cache C1、C2 中修改了 A1、A2。MESI 一致性协议确保了内存和 cache 的一致性

在 cache 一致性的逻辑控制下，可以避免这种现象。例如，MESI 协议中的 4 个字母表示 cache 行的 4 种可能状态：

- M 修改（modified）：此 cache 行在 cache 中被修改，且在其他 cache 中没有其他备份。只有写回内存后，内存才表示为最新状态。
- E 专有（exclusive）：此 cache 行只从内存中读取没有被修改，但是只存在这一个 cache 中。
- S 共享（share）：此 cache 行只从内存中读取没有被修改，但是在其他 cache 中可能还有备份。
- I 无效（invalid）：此 cache 行的数据无效。在正常情况下，只有当该 cache 行处于共享状态情况下，并且某一个处理器要求独有该数据时才会发生。

图 4-2 中描绘了事件的顺序。如果一个 cache 行处于 M 状态，而其他 cache 需要读取该行的最新数据，当需要写回内存时怎样才能被其他 cache 注意到呢？同样 cache 行处于状态 S 或者 E 时，如果其他 cache 需要独享这个数据时，该行必须置为 I 状态（invalidated）。在小型系统中可以用总线窥探（snoop）来实现：当 cache 有序发布通知时，原始发起该通知的 cache 通过系统广播一致的 cache 行地址，所有其他 cache 窥探总线，并作出相应反应。虽然该方案可以简单实现，但是存在着一个重要的缺点，当在系统总线上广播地址时，容易被污染，导致有效的存储带宽减少。使用一个专门的网络可以减少带来的影响，但也并不总是可行。

98 在大型 ccNUMA 机器中，目录协议通常会作为一个更好的替代方案：芯片和内存接口等总线逻辑跟踪系统中 cache 行的位置和状态。这虽然会占用一小部分主存或 cache，但是 cache 行状态的改变只传给需要该数据的 cache，这大大减少了通信网络中为确保一致性而进行的数据传输。目前甚至使用工作芯片组实现“窥探过滤器”来达到相同的目的。

如果相同的 cache 行被不同的处理器频繁修改，则通信网络会严重伤害应用程序的性能。7.2.4 节将会介绍在用户程序中怎样避免伪共享的情况。

4.2.2 UMA

UMA 系统最简单的实现是一个双核处理器，两个 CPU 在同一个芯片上共享一条存储路径。高性能计算中在计算节点上使用多个单核或者多核芯片也是很常见的现象。

图 4-3 中，两个单核处理器都有自己的插槽，通信和访存都使用同一个前端总线（FSB），CPU 中已经内建了所有的仲裁协议。图中的 chipset（芯片组，通常是“北桥”）负责驱动存储模块和连接其他部分的节点，如 I/O 总线。然而这种设计已经过时，现代系统已不再使用。

图 4-4 中，两个双核芯片与 chipset 相连，每一个核都有独立的 FSB。chipset 能增强 cache 的一致性，也连接着内存。原则上，这样的系统能够使 chipset 到内存的带宽与前端总线的总带宽相匹配。每个芯片上有一个共享的 L2，每个核都有单独的 L1。核、cache 和插槽的分布使系统存在各向异性，即一个核与其他核的“距离”取决于是否在同一插槽上。大规模的多核处理器拥有多级 cache 组，这使情况变得更复杂。参考 1.4 节关于共享 cache 和各向异性结果的更多信息。 99

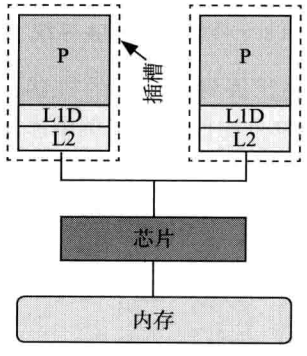


图 4-3 两个单核 CPU 组成的 UMA 系统，它们共享一条前端总线

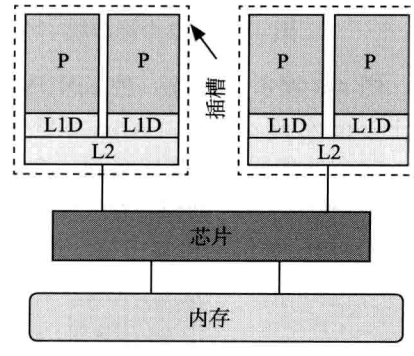


图 4-4 两个双核 CPU 组成的 UMA 系统，两条前端总线分别连接到芯片组

UMA 系统的主要问题是当插槽（或者前端总线）的数量超过了限制就会产生带宽瓶颈。像图 4-3 中的设计，存储总线通常一次只能传输一个数据给一个 CPU（如今的所有多核芯片都会受到带宽的限制，但将来可能会发生一些变化）。

为了保持内存带宽和 CPU 数量的可扩展性，可以内建无堵塞开关来建立插槽和存储模型间的点对点连接，如图 4-4 中的芯片。但是随着插槽的增加，高聚合的带宽将是一笔昂贵的费用。本书写作时，最大的具有可扩展的带宽 UMA 系统（NEC SX-9 向量节点）只有 16 个插槽。这个问题无法解决，除非放弃 UMA 原则。

4.2.3 ccNUMA

在 ccNUMA 结构中，多个处理器核和本地连接内存一起称为局部域（Locality Domain，LD）。其中的内存能被最高效的方式访问，即不需要连接任何网络。多个局部域通过网络连接，允许处理器透明地访问其他处理器的内存。从这个意义上可以说，一个局部域可以看作是一个 UMA 群（building block）。整个系统仍然共享内存，运行着同一个 OS 实例。尽管 ccNUMA 提供可扩展带宽能扩展到很多个处理器，但是在 HPC 集群中广泛使用一种比较便宜的 2 或 4 个插槽的节点模式（参见图 4-5）。在图 4-5 的例子中，有两个局部域，每一个局

部域中有四个处理器，它们分别有着私有 cache，还有一个通用接口连接着本地内存。两个局部域由高速网络连接。超传输（HyperTransport，HT）和快速通道（QuickPath，QPI）是 AMD 和 Intel 目前比较青睐的技术，但是也存在一些其他连接技术。由于插槽能够直接驱动内存，因此内部插槽能直接访问 cache 一致性内存，使得单独的接口芯片被淘汰。从程序员的观点来看这种机制也是透明的：硬件直接处理所有需要的协议。

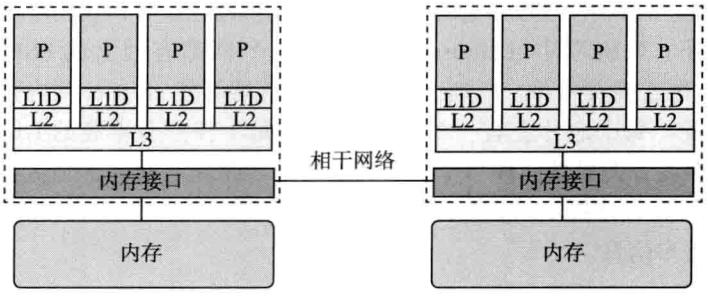


图 4-5 有 2 个局部域（每个插槽上面一个局部域）和 8 个核的 ccNUMA 系统

100 图 4-6 展示了另外一种 ccNUMA 设计，它可以灵活地扩展到大规模的机器，常被用在基于 Intel 处理器的 SGI Altix 系统中，在单个地址空间和单个操作系统实例中连接着上千个核。每个处理器插槽都连接到一个通信接口（S），该通信接口提供内存访问并连接着专用的 NUMALink（NL）网络。NL 网络依赖路由器访问非局部存储。由于 HT 和 QPI 技术，允许 NL 硬件透明地访问机器中整个地址空间中的所有核。

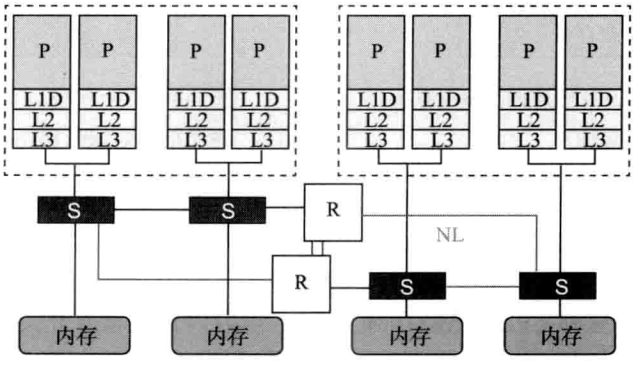


图 4-6 一个 ccNUMA 系统（SGI Altix），有四个局部域（LD），每个 LD 都包含一个双核的插槽。LD 通过路由器使用 NUMALink（NL）网络连接

图中只展示了四个插槽，但是通过多级路由器可以把规模扩大到几百个 CPU。需要注意，在数据连接（通信接口、路由器）中插入的每一个硬件都会增加延迟，这会造成系统中访问时间的不均匀。而且在大型系统中，提供相当于直接连线的速度和远程非堵塞内存访问非常昂贵。由于这些原因，大型超级计算机和成本要求较高的小型集群通常由共享内存的集群（通常是 ccNUMA 种类）通过一些不具有 ccNUMA 能力的网络连接而成。详细细节请参考 4.3 节和 4.4 节。

在所有的 ccNUMA 设计中，网络连接必须与本地存储有相同数量级的带宽和延迟。在

所有的现代系统中，如果访存不限制在局部域内，那么即使一个很小的惩罚系数也能很大程度地损害应用程序的性能。在 ccNUMA 高性能软件中，局部域问题是影响性能的两个障碍之一，甚至在只运行一个串行程序的 ccNUMA 机器上也会发生。第二个问题是潜在的竞争。如果不同局部域中的两个处理器需要访问同一个局部域中的内存，就存在内存带宽的竞争。即使网络无堵塞、性能在带宽和访存延迟限制以内，竞争也可能发生。这两个问题可以通过仔细观察并安排应用程序数据的访存模式、限制每个处理器只能访问其局部域中的数据存储等方法避免，第 8 章将会详细介绍。

尽管 I/O 传输相对于内存带宽较慢，但是也有些高速网络能在计算节点之间能达到 GB 的带宽。在廉价的 ccNUMA 系统中，I/O 接口通常只连接到一个 LD 上。如果数据写到了 101
“错误”的局部域，I/O 驱动将忽略所有的 ccNUMA 约束而把它复制到最优的地址空间，但这将减少 4 倍的有效带宽（如果写分配可以被避免，则是 3 倍，见 1.3.1 节）。在这种情况下，再贵的互联硬件也是浪费。真正可扩展到 ccNUMA 设计中，通过机器中的分布式的 I/O 连接和 ccNUMA-aware 驱动可以绕开这个问题。

4.3 分布式存储计算机

图 4-7 是一个分布式内存并行计算机简图。每个处理器 P 连接一个独有的局部内存，即其他 CPU 不能直接访问该内存。但是现在已经没有以这种层次结构实现的分布内存系统了，这个草图只是被当作一个编程模型看待。一方面，由于价格或性能方面的原因，当今所有的并行机中最流行的是 PC 集群，由一批具有 2 个或更多 CPU 的共享内存的“计算节点”组成（见下一节）。另一方面，从“分布式存储程序员”的角度来看也不是这样的，在纯共享内存的机器上使用分布式存储方法编程也是可能的（也十分常见）。

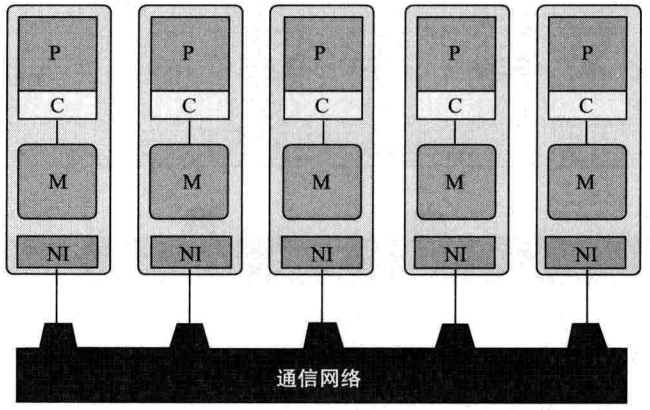


图 4-7 分布式并行计算机的“程序员简化版”或“编程模型”：处理器中运行着独立的进程，通过网络接口（NI）在网络上通信，虽然处理器可能储存于共享内存中，但处理器不能直接访问其他处理器的内存（M）

每个节点至少包含一个网络接口（NI）连接到通信网络。每个 CPU 上运行一个串行程序，该程序能够通过网络与其他 CPU 上的程序进行通信。可以预想到，在一个共享内存的并行计算机上，几个处理器同时解决一个问题是很容易的，但是在分布式存储计算机上没有远程访存功能，需要通过在处理器之间的消息传递来合作解决一个问题。第 9 章将介绍主流

[102] 的消息传递标准：MPI。虽然消息传递比共享内存编程模型更复杂，但是从总体上来看，大规模的超级计算机是分布式内存系统的变体。

这里列出的分布式内存架构也称为无远程内存访问模型（No Remote Memory Access，NORMA）。有些厂商提供库和硬件支持一定的远程访存功能，甚至是在分布式内存机器上。但是这种特性是与厂商强相关的，没有被广泛接受的标准，详细的介绍已超出本书的范围。

互联网络的选择有很多。最简单的情况可能是标准交换式以太网了，但也出现了一些更先进的技术可以很容易达到千兆以太网十倍的性能（网络的基本性能特点请参考 4.5.1 节）。5.3 节将看到，网络的布局和速度对应用程序的性能有很大的影响。最受欢迎的设计是由一个非阻塞“线速”网络组成，该网络能够在 N 个节点之间不产生任何瓶颈地转换 $N/2$ 个连接。尽管一些几十到上百个节点的小型系统一应俱全，但是其中非阻塞交换结构的大型装置是一个巨大的浪费，而如果摈弃这些装置，那么当所有节点同时要求通信时则会出现瓶颈，所以这通常要做一些折中。详细的网络拓扑请参考 4.5 节。

4.4 混合型系统

正如前面所提到的，大规模并行计算机并不是纯粹的共享内存类型也不是纯粹的分布式存储，而是两者的结合，即用共享内存的结构通过快速网络连接而成。由于网络增加了其他
[103] 的通信特点（如图 4-8 所示），这使得总体系统设计与多核或者 ccNUMA 节点设计比较起来更加具有各向异向性。混合型的概念会使尽可能多的基础设施可以共享，具有很好的性价比，例如通过两个插槽建立一个共享内存节点比和两个节点各自拥有的一个插槽的系统明显便宜。而且更多的核和插槽共享一个网络连接，这也减少了网络的开销。

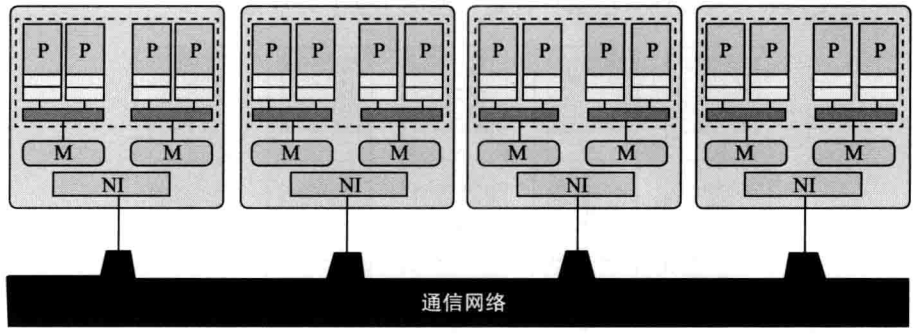


图 4-8 典型的具有共享内存节点（ccNUMA 类型）的混合型系统。双插槽结构具有很好的性价比优势，并应用在许多商业集群上

目前双槽结构是便宜的商品集群最好的选择，即使用标准的组件而不是专门为 HPC 设计的组件建立系统。根据系统上具体运行的应用程序，对于这种结构，每个核可用的网络带宽减少会导致程序的性能有所限制。此外它本身并不清楚怎样高效地利用核、cache 组、插槽和节点之间的复杂结构。唯一普遍的共识是最优的编程模型高度依赖应用程序和系统。第 11 章将会具体讲解这种层次结构的编程方法。

具有上述层次结构的并行计算机也称为混合型（hybrid）。这个概念更加通用，通常用来分类在不同硬件层次上是否具有混合编程模型的系统。这种系统的一个突出例子是由节点组成的集群，节点除了“通常”的多核处理器外，还有额外的加速硬件，从应用程序特

定的卡到 GPU（图像处理器）、FPGA（现场可编程门阵列）、ASIC（特定用途集成电路）、协处理器等。

4.5 网络

5.3.6 节将看到通信的开销对应用程序性能的巨大影响。网络连接着“执行单元”、“处理器”、“计算节点”等在很大程度上影响应用程序性能的特性。市场上有很多种类的网络连接技术和拓扑结构，其中一些私有而一些开放。本节将简单阐述应用在 HPC 领域的几种不同网络的拓扑结构和性能。我们将试着独立于具体应用和程序模型来讨论，重点讨论应用在 104
分布式内存、共享内存和混合型系统上的网络。

4.5.1 网络的基本性能特征

正如前面所提到的，并行计算机上有很多种网络可选择。最简单也最便宜的应该是千兆以太网，它将满足多数应用程序的要求，但是对于有快速通信需求的并行程序来说还是太慢。在本书写作时，分布式内存特别是在商业集群上的网络的主要选择是 InfiniBand。

1. 点对点连接

不管底层硬件是什么，单个点对点连接的通信特点可以通过以下简单模型来描述：假设传输 N 字节的消息，总的传输时间由延迟和数据流组成，

$$T = T_\ell + \frac{N}{B} \tag{4-1}$$

B 是最大（渐近）网络带宽，单位为 MB/s，有效带宽是：

$$B_{\text{eff}} = \frac{N}{T_\ell + \frac{N}{B}} \tag{4-2}$$

注意在通常情况下，T_ℓ 和 B 依赖于 N 的长度。一个典型例子是如图 1-18 所示的共享 cache 的多核处理器，在同一个插槽的两个核上，消息传递的延迟和带宽明显依赖于消息的长度是否符合共享 cache 行的长度。我们暂时忽略这个影响，但在理解消息传递的优化细节上这是至关重要的，我们将在第 10 章继续讨论这个话题。

在测量延迟和有效带宽时常用 PingPong 基准测试集。这个代码在运行在不同处理器上（也有可能是不同的节点，参见图 4-9）的两个进程之间发送并接收一个长度为 N 字节的消

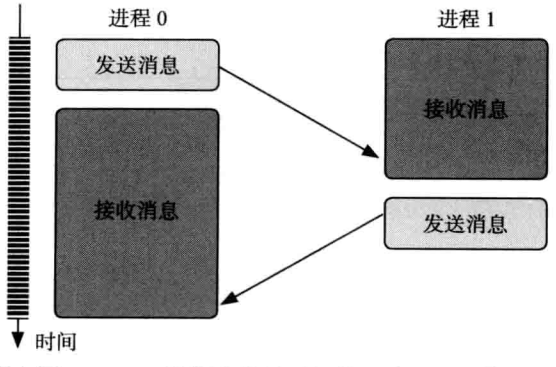


图 4-9 在两个处理器之间 PingPong 数据交换的时间线。处理器 0 传送长度为 N 字节的消息到处理器 1，然后再传回

息。伪代码如下所示：

```
myID = get_process_ID()
if(myID.eq.0) then
  targetID = 1
  S = get_walltime()
  call Send_message(buffer,N,targetID)
  call Receive_message(buffer,N,targetID)
105
  E = get_walltime()
  MBYTES = 2*N/(E-S)/1.d6      ! MBytes/sec rate
  TIME   = (E-S)/2*1.d6        ! transfer time in microsecs
                               ! for single message
else
  targetID = 0
  call Receive_message(buffer,N,targetID)
  call Send_message(buffer,N,targetID)
endif
```

接下来讨论不同的 N 对于带宽的变化，带宽单位 MB/s。现实中经常使用合适的消息传递库，例如第 9 章介绍的消息传递接口（Message Psssing Interface，MPI）。以下所示的数据是用标准的“Intel MPI 基准测试集”（Intel MPI Benchmark，IMB）获得的 [W 124]。

在图 4-10 中，对式（4-2）在千兆以太网上测量出的真实参数数据进行拟合以建立模型，这个简单的模型能够很好地描述总体特点：我们发现消息长度较小时占用的带宽非常低，因为主要的传输时间是延迟；而消息长度很大时，延迟几乎可以忽略但带宽却将近饱和。拟合的参数为千兆以太网上合适的值。然而令 $N = 0$，延迟时间即为传输时间（见图 4-10 中的插图）。但是不能精确地拟合 T_ℓ，接下来的章节会给出更详细的解释。

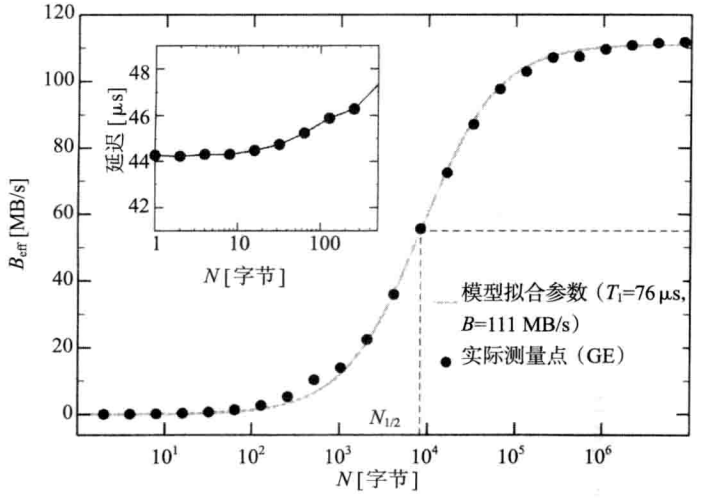


图 4-10 在千兆以太网上，式（4-2）中的模型在不同消息长度时有效带宽的参数拟合。不能精确拟合出的 T_ℓ 值（文中有解释）。$N_{1/2}$ 是达到饱和带宽一半时的消息长度（虚线）

延迟是由数据链路的物理参数设定的，与带宽限制比较起来，有以下几个特点：

- 所有的数据传输协议都会有一定的开销，例如像消息头一样数据管理结构。
- 有些协议（例如网络层使用的 TCP/IP 协议）定义了一个最小消息长度，所以即使传输单个字节的消息，传输的最小帧的长度也大于 1 字节（$N > 1$）。
- 消息传输的初始化是一个复杂的过程，根据网络协议的复杂度不同涉及多个软件层，每一个软件层都会增加延迟。

❑ 集群中经常使用的 PC 硬件，一般没有对低延迟的 I/O 进行优化。

事实上，高性能的网络通过减少以上各点带来的影响来改善延迟。供应商提供的轻量级的协议、优化驱动、通信设备直接与处理器总线相连等措施都已经用于减少延迟。

但是模型的参数拟合与实际情况还是有出入（见式（4-2）），毕竟随着消息长度的变化，有效带宽和延迟会有很大的变化。例如消息长度为 8 个数量级的差距，并且延迟为主体的系统中（小的消息长度模式）有效带宽要比长度大的消息少 3 个数量级以上。而且，T_ℓ 和 B 分别在不同的消息长度下才能拟合出最佳的参数。正是由于这些原因，在图 4-10 中千兆以太网延迟的 PingPong 测试失败了。因此，测试应用程序的参数模型应该在不同的消息长度 106
（至少取两个极端长度）下测量。图 4-11 是 DDRInfiniBand 网络测量 PingPong 测试集的结果。为了更好地判断拟合的质量，在图 4-11 中将两个坐标轴写成对数形式。分别对小消息长度和大消息长度进行测量，严格拟合形成了点虚线和虚线，前者能够获得更好的 B 的估计值，而后者能够得到更精确的 T_ℓ。实线是使用拟合函数结合两个参数（式（4-2））拟合成的，对中间消息长度测量的结果。造成参数拟合结果有误差的原因有很多，通常是由于给定了消息长度，消息传递和网络协议层次在不同的缓冲算法中转换（参见 10.2 节）或者由于消息超过了大小的限制被分为更小的块等。

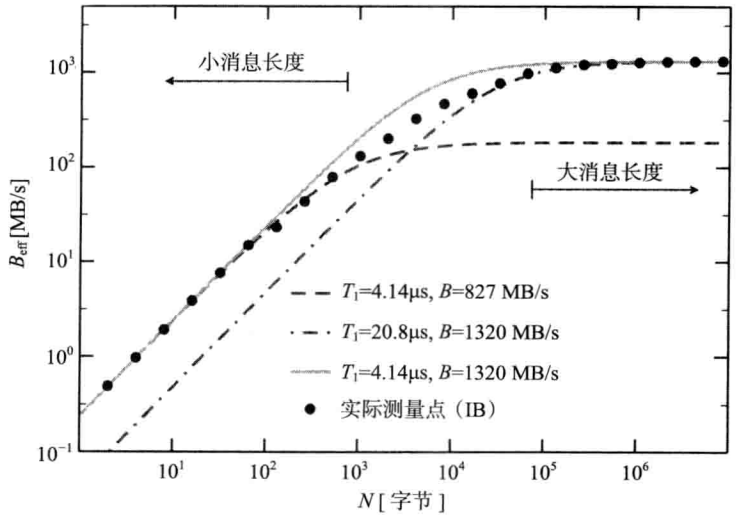


图 4-11 DDR InfiniBand 网络上，式（4-2）中的模型在不同消息长度时有效带宽的参数拟合。分别列出了最合适的参数：渐近带宽（点虚线）、延迟（虚线），实现是使用了结合两者的拟合函数（实线）

虽然饱和带宽 B 可能非常高（有些系统能使节点间的网络带宽可以与处理器本地内存之间的带宽相媲美），但是许多应用程序仍然工作在带宽图中延迟占主要部分的区域。为了量化这个问题，引用了 $N_{1/2}$，表示当 $B_{eff}= B/2$ 时消息的长度（见图 4-10）。在这个模型中（式（4-2）），$N_{1/2} = BT_\ell$。这可以用来解释以因子 β 增加最大网络带宽是否真的有利于所有消息。当消息的长度为 N 时，对带宽的改善为：

$$\frac{B_{eff}(\beta B, T_\ell)}{B_{eff}(B, T_\ell)} = \frac{1+N/N_{1/2}}{1+N/\beta N_{1/2}} \tag{4-3}$$

所以当 $N = N_{1/2}$，$\beta = 2$ 时，只增加 33%。同样以因子 β 减少延迟是一样的结果。因此，对于所有的应用程序，这是改善延迟和带宽以使互联更有效的理想的模型。

2. 对分带宽

上述简单的 PingPong 算法并不能准确描述“总体”饱和带来的影响：如果网络并不是完全非堵塞结构，当所有节点同时传输或者接收数据时，聚合带宽即所有点对点连接的有效带宽的总和会低于理论值。这会严重减少多 CPU 系统上应用程序的性能和总吞吐量。对分带宽（bisection bandwidth）B_b 可以用来度量整个网络上的最大聚合通信能力。把整个网络切分为两个相等大小的部分（图 4-12 中的虚线）时，必须去掉的连接边的带宽总和的最小值即为等分带宽。在混合系统中，每个核可获得的带宽是一个更重要的度量方法，即等分带宽除以核的数量。每个核的等分带宽下降是多核传输中一个额外的不利影响因素。

107
~
108

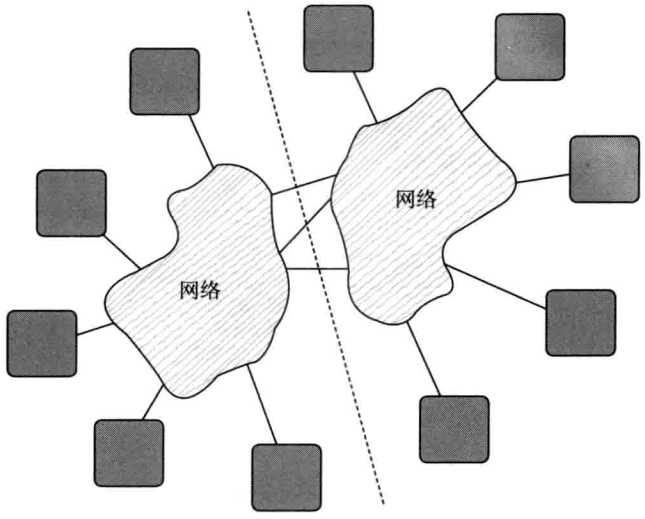


图 4-12 等分带宽 B_b 是当等分系统分成两个部分时至少必须去掉的连接边（图中是 3）的带宽总和

4.5.2 总线

总线是一种共享介质，但同一时刻只能被一个通信设备使用（图 4-13）。需要一些合适的硬件机制以阻止发生冲突（两个以上的设备同时传输）。总线在计算机系统中非常常见，它很容易实现、延迟少，而且能提供必要协议需要的现成硬件。PCI（Peripheral Component Interconnect）总线是典型的例子，在许多商用系统中用来连接 I/O 设备。在目前一些多核设计中，总线与主存在同一块芯片中单独连接着 CPU 芯片。

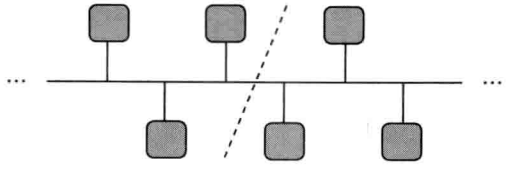


图 4-13 总线网络（共享介质）。同一时刻只有一个设备能使用总线，对分带宽与节点的数量无关

总线最重要的缺点是堵塞。所有设备共享一个恒定带宽，意味着连接的设备越多则平均带宽越低。在工艺设计上，为大型系统设计更快总线时，电容和电感负载限制着传输的速

度。此外，总线很容易由一个局部问题而影响到所有设备。在高性能计算领域，高速通信时使用的总线往往限制在处理器或者插槽层上，或者只针对专门的网络使用。 109

4.5.3 交换网络和胖树网络

交换网络（switched network）中把所有的设备细分为组。组中的所有设备以星型形式连接着一个中央网络设备（交换机），交换机互相连接或者通过其他的交换机层次连接。在这样的网络中，两个通信设备的距离通常由“跳”（hop）来描述，即消息传递时需要经过交换机的数目。因此一个多交换机的层次结构必须考虑延迟。任意两个互联设备之间跳数的最大值即为网络直径，同一个总线系统（参见 4.5.2 节）中网络直径只有一个。

一个交换机或者可以单独支持一个完全非堵塞操作，即每对端口可以同时使用它们之间的全部带宽，或者也可以像总线那样带宽会被限制。完全无堵塞交换的一种实现方法是使用开关（crossbar，见图 4-14），这样的结构可以被连接并串联成一种胖树交换层次，这样既可以在整个系统中保持非堵塞（如图 4-15），也可以减少树根节点的连接以“定制”合适带宽（如图 4-16 所示）。在后者的结构中，每个节点的对分带宽会少于每个端口叶子交换机带宽的一半，而且即使静态路由本身没有任何问题也可能发生竞争。网络基础设备应该具有平滑流量（动态或者静态）的能力，否则一些较重负载的点对点连接会比一些轻载连接更快。另一方面，任意两个计算节点之间的最大延迟通常只与交换层次的数量有关。

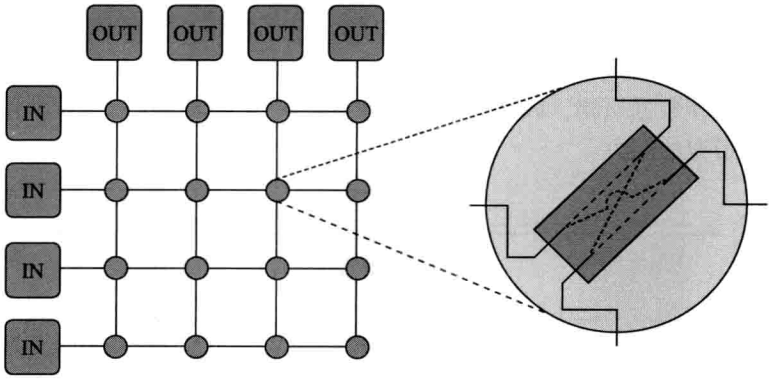


图 4-14 一个完全无堵塞的二维交叉胖树网络。每一个圆圈代表从“IN”到“OUT”的两个设备之间的一个可能的连接，所以圆圈形成了一个 2*2 的交叉网络。整个回路可以作为一个 4 端口的非堵塞交换机

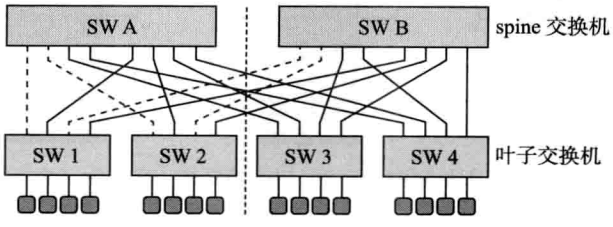


图 4-15 一个完全非堵塞满带宽的胖树网络，具有两层交换机结构。连接计算元素的交换机称为叶子交换机，而顶层交换机形成了结构的 spine

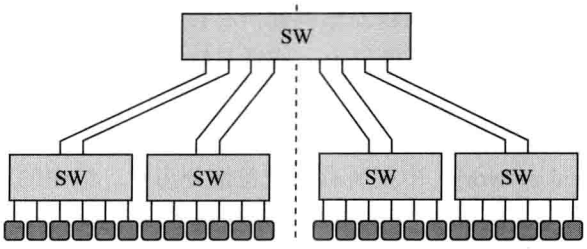


图 4-16 由于 spine 的通信连接出现“1：3”的现象会导致胖树网络出现瓶颈。使用单个 spine 交换机，由于只可能有 4 个非堵塞连接对，所以与图 4-15 比较起来对分带宽只有一半，而每个计算节点的对分带宽会更少

110 由于建立上千个计算节点的完全非堵塞交换结构非常昂贵，而交换机所需的硬件和布线比较容易做到，所以上文中的后一种是大型系统常用的选择。此外，考虑到聚合带宽，网络将变得很不均匀——根据应用程序的实际通信需求，任务在整个系统中的确切位置对整体性能非常重要：如果一组任务只使用一个叶子交换机，则它们都会乐于使用完全非堵塞通信而不论是否出现瓶颈（见 4.5.4 节的转换方法，建立很大的高性能网络可以避免这种问题）。

然而像图 4-15 那样的完全非堵塞交换结构仍然可能出现瓶颈。如果使用静态路由，即若计算节点之间是“硬布线”连接，也就是说在两个节点之间有且只有一条数据通路（交换
111 开关遍历的次序），可能出现 spine 交换机端口的不平衡使用，当负载很高时就会导致冲突（参考图 4-17 的例子）。如今的许多商业交换机产品使用静态路由表 [O 57]。自适应路由可以根据网络负载选择数据通路，因此能够避免冲突。对于所有的通信模式，只有自适应路由有充分利用对分带宽的能力。

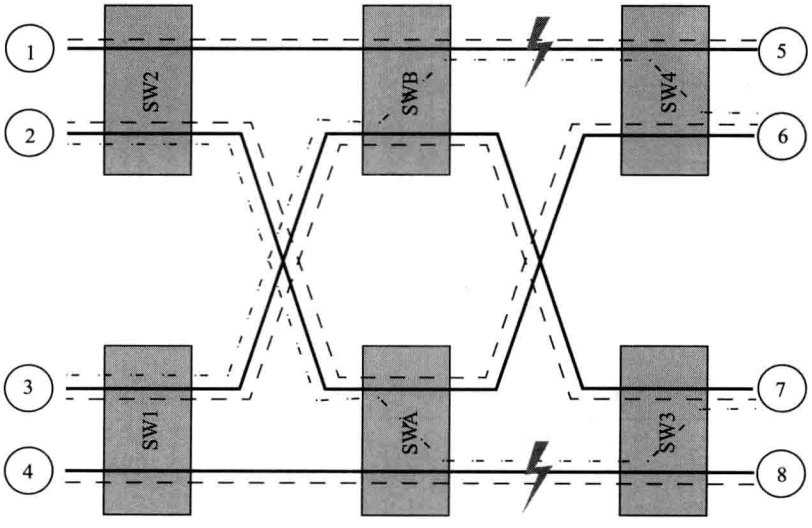


图 4-17 如果使用静态路由，即使在无堵塞的胖树交换结构（实线为网络布线）中，也并不是所有的 $N/2$ 个点对点连接都允许无碰撞操作。开始时的无冲突连接模式如图中虚线所示，当 2↔6 和 3↔7 的连接突然变为 2↔7 和 3↔6 之后（点虚线），就会发生冲突，假设 1↔5 和 4↔8 的连接没有重新选择路由路线

4.5.4 Mesh 网络

胖树网络交换结构在大型系统中的规模会受到限制，特别是性价比很低。元器件和大量布线的成本让人望而却步，因为某些方面经常会做一些牺牲，例如减少每个计算节点的对分带宽。为了克服这个缺点，一些大型 MPP 机器 像 IBM Blue Gene[V114，V115，V116] 和 Cray XT[V117] 等经常使用 Mesh 网络，通常是多维立方体（超立方体）的形式。每个计算节点位于笛卡儿网格的交点处。通常超立方体的边界被连接包裹起来形成一个环面拓扑结构（图 4-18 是一个二维环面的例子）。不邻接的元素之间不会直接相连。通过这样的系统路由数据时经常使用特殊的 ASIC（Application Specific Integrated Circuit，应用专门集成电路）来完成，它能够考虑到所有的网络流量，而且尽可能地绕过 CPU。网络直径是所有 3 个笛卡儿方向中系统大小的总和。

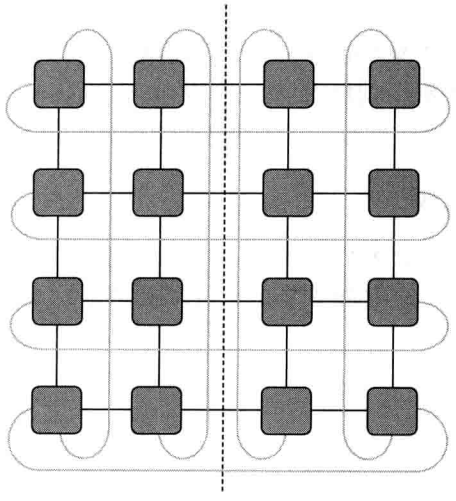


图 4-18 二维环面（正方形）网络。本例中的对分带宽规模为 $N^{1/2}$

112

在所有维度中随着系统的增大，对分带宽的规模并不是线性增加而是满足 $B_b(N) \propto N^{(d-1)/d}$（其中 d 为维度），当 N 很大时，$B_b(N)/N \to 0$。另外，最大延迟的规模为 $N^{1/d}$。尽管第一眼看起来这些属性并不让人兴奋，但是对于主要是近邻之间通信的大型应用程序来说，环面拓扑结构是一个受欢迎而且很划算的选择。如果链路的最大带宽远远大于单个计算节点能够流入到网络中的带宽（它的注入带宽），那么就有足够的带宽能满足更苛刻的通信模式（如 Cray XT 上大规模的并行计算机 [V 117]）。立方 Mesh 网络的另一个优点是布线很少，而且大部分的连接线都很短。与胖树网络比起来，在带宽和延迟上有很多不同，但是可以预见，并行工作时，计算节点之间位置比较靠近（同在一个立方体区域）。此外，由于成本或者管理方面的原因，当某一个节点的对分带宽突然减少时也不会对系统的大小产生太大的影响。

在一些小型具有 ccNUMA 能力的共享内存系统中，简单的 Mesh 网络常用于局部域之间的连接。图 4-19 是一个使用超传输的 4- 插槽系统的例子，由于两个 HT 连接用做 I/O 连接：右边两个局部域中的任何通信都会由另外两个局部域中的某一个产生额外的一跳，所以这些节点实际上实现了一个异质性的拓扑结构（在插槽内部的延迟方面）。

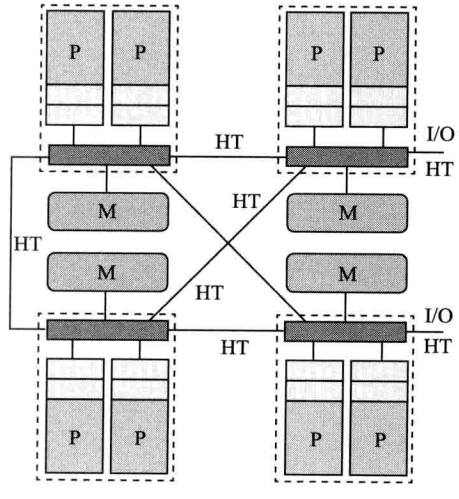


图 4-19 使用 HT Mesh 网络的 4- 插槽 ccNUMA 系统。每一个插槽有 3 个 HT 连接，要适应 I/O 通信网络必须为异质性，而且要使用全部的 HT 端口

4.5.5 混合网络

如果某个网络有上述至少两个网络混合连接而成，则称为混合网络。由于节间连接往
113 往是总线（多芯片中）或者简单的 Mesh 网络（像 HyperTransport 或者 QuickPath 一样具有 ccNUMA 能力的立方体网络），所以像图 4-8 一样的共享内存节点实现的集群就是一个混合网络，即使它的内部网络并不是混合的。规模很大时，在小数量的节点组之间使用立方体拓扑然后再使用胖树网络组织起来可以有效减少纯立方体 Mesh 网络的对分带宽问题。

习题

4.1 构建胖树网络结构问题。静态路由的胖树网络结构，如果链路上 spine 交换机的输入与输出比为
114 2：3(参考图 4-16，它的比例为 1：3)，则会发生什么问题？

并行性基础

开始真正进行并行编程之前需要了解一些并行化的基本规则，这涉及判断可并行化的操作，甚至是更为重要的性能瓶颈。最常见的错误之一是认为并行程序在更多的硬件上会运行得更快。其实，由于超级计算机的并行执行瓶颈等问题，每年浪费数以百亿计的处理器小时。

本章首先介绍常用并行化策略并对其分类，然后在理论层次上研究并行性，包括几种简单的用来推断影响并行程序性能因素的数学模型。虽然这些模型的可应用性和可预测性有限，但是它们提供了与具体并行编程模式无关的并行化机理。实践编程标准将会在接下来的几个章节进行介绍。

5.1 为什么并行化

并行性概念在大规模并行机到多核笔记本电脑等不同平台中随处可见，然而由于单核处理器能够满足其性能要求，所以许多科学计算用户并没有真正编写并行程序。如果单核处理器不能满足其要求，那么主要的两个原因如下：

- 单核处理器太慢而不能在合理的时间内完成任务，此处合理的定义随着环境的变化而改变，例如一天通常是一个可接受的范围。根据需求的不同，几个小时可能更为合理。
- 内存需求不能由单一的系统满足，例如大规模程序（高精度、多物理、多粒子等要求）需要的内存量十分巨大。

第一个问题在多核处理器发展趋势下应该是不能避免的。在并行机出现之前的很长时间内，第二个问题通过称为核外（out-of-core）的技术解决，即绝大部分数据集存储在大规模存储设备上，只有在需要时才被加载，大多数情况下这对性能影响较小。然而，峰值性能和 I/O 带宽（和延迟）的差距甚至比内存处理器间的差距增长得更快，所以未来核外技术也许并不能用于串行计算。当前并行计算机通过并行文件系统提供高速 I/O 资源，当数据来自不同数据源时能达到最高性能。

以下各节对各种并行性方法逐一进行介绍。

5.2 并行性

编写并行程序的第一步是分析并行算法中的并行性。不同的并行性可以通过不同的并行化方法实现。本节对常见并行化方法进行介绍，期望可以启发读者阅读其他并行化方法的优秀文献。Mattson 等 [S6] 对并行编程模式进行了深入的介绍。本书只介绍多核或多节点上的并行性分析方法。第 1 章和第 2 章介绍了细粒度并发性例如超标量处理器或 SIMD 处理器的分析方法。

5.2.1 数据并行性

科学计算中许多问题都涉及大量实验数据的处理，如果计算可以被并行执行，即多处理器在不同的数据上进行并行计算，则称为数据并行。事实上这是 MIMD 类型的计算机上最为常用的科学计算并行化概念。它也被称为单程序多数据（Single Program Multiply Data，SPMD）模式，即多个处理器以独立指令指针执行同样的代码，注意这和 SIMD 并行概念的区别。

1. 实例：中粒度循环并行性

通过循环和嵌套循环处理数组数据是大多数科学计算程序的主要组成部分。典型的例子是线性代数操作中的向量或矩阵，例如标准 BLAS 库中的代码 [N50]，通常每个数组元素的处理是相互独立的，因此是共享内存的多个处理器上并行执行的典型对象（见图 5-1）。这种并行计算方法称为中粒度的原因在于处理器间的任务分配是可以调节的，甚至可以分配单独
116 一个元素给某个处理器。与图 5-1 所示的方法截然相反，也可以进行交替模式的分配，即将所有奇数（偶数）索引的元素分配给 P1（P2）。

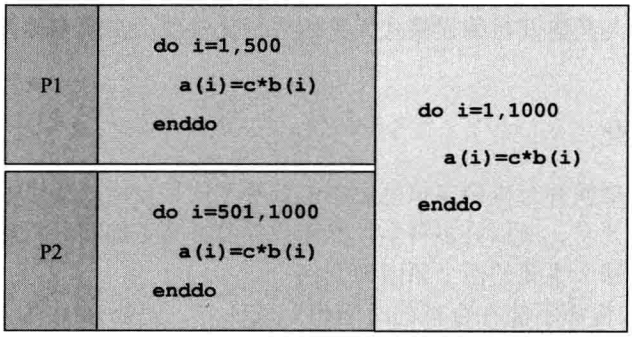


图 5-1 中粒度并行性实例，循环中的所有迭代分布到两个处理器 P1 和 P2 上并行执行（共享内存）

OpenMP 是一种基于指令和 API 的编译器扩展，支持循环代码的数据并行，第 6 章对 OpenMP 进行介绍。

2. 实例：通过区域分解进行粗粒度并行化

物理过程（例如流体、机械压力、量子场等）模拟通常利用一个非常简单的图像对实际问题进行刻画，其中有一个计算域，例如一定规模的流体，被表示成具有一系列离散位置的网格（例如 3.3 节中介绍的 Jacobi 算法），这些网格不一定是笛卡儿坐标系的，而通常是适应所使用算法数值的限制，模拟通常就是为了计算这些网格点上变量的数值。对计算进行划分的一种直接方法是将网格分配到不同处理器上，即区域分解方法。例如考虑一个二维 Jacobi 解法，它在 $n \times n$ 网络上更新物理变量。区域分解方法将网格划分为 N 个子域并分配到 N 个处理器上，例如，如果按 y 轴方向将网格划分为条状（即代码清单 3-1 中的索引 k），每个处理器将对本地的子网格扫描一遍，每时间间隔 T_1 更新数组。在共享内存并行计算机上，所有域中所有的网格节点在下一次处理器同步之前都可以进行更新。但是，在分布存储系统上，更新域中边界节点需要一个或者多个相邻节点处理器的信息，因此在进行下一次域更新之前，所有节点需要与相邻子网格节点处理器进行通信，为了存储这些交换的边界信息，每个域必须存储一些额外网格节点（称为 halo 或者 ghost，见图 5-2）。交换信息后，每个域可以进行下一次迭代，因此整个并行算法等价于串行算法。9.3 节展示了使用 MPI 进行算法实
117 现的细节。

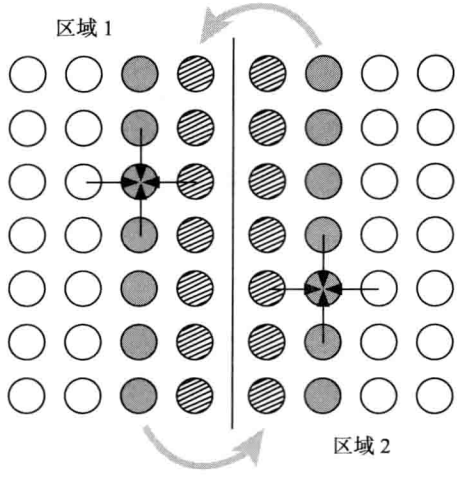


图 5-2 在分布式存储系统上使用 halo（ghost）层进行子域间通信的 Jacobi 迭代。每个域的本地更新步骤完成后，边界点（实心部分）需要被复制到相邻区域（阴影部分）

因为多个因素影响最优选择，所以如何对整个网格进行子域划分是一个十分困难的问题。首先并且最重要的是计算量需要平均地划分到所有子域，以防止某些线程等待而其他一些线程还在进行更新计算，即负载均衡问题（见图 5-5 和 5.3.9 节）。在解决负载均衡问题后，需要考虑如何降低通信开销。需要通信的数据规模正比于区域划分的规模。比较图 5-3 中的两种 $n*n$ 网格二维区域分解，第一个条状划分方法的通信开销是 $\mathcal{O}(n(N-1))$，但是最优的分解方法为二维划分，通信开销为 $\mathcal{O}(2n(\sqrt{N}-1))$。因此对于足够大的处理器数目 N，最优分解方法的通信开销为 $\mathcal{O}(2/\sqrt{N})$，实际是否会与理论分析有差距还依赖于其他因素，例如问题规模。通信是提高程序性能必须要考虑的开销，除非有其他原因，否则实践中需要尽量降低边界区域，10.4.1 节提供了更为详细的讨论。

注意通信开销的计算与数据依赖的局部性密切相关，通信开销随着信息传递路径的增加而线性增加，例如，为了得到某个量相对于坐标的一阶或二阶导数，仅需要相邻子域信息，在图 5-3 中只需要宽度为 1 的通信层，对于更高阶的导数需要更宽的通信层，如果有类似 Coulomb 势一样的大范围交互（1/ 距离），需要传递所有计算域，这样通信性能就是整个程序的瓶颈，因此对于这种情况，区域分解方法不再可行，需要寻找其他并行策略。

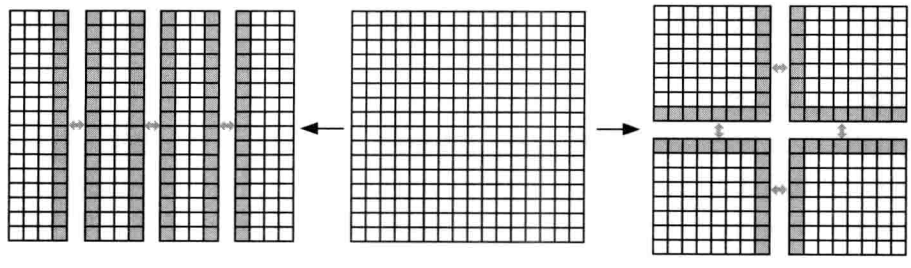


图 5-3 二维 Jacobi 区域分解算法，需要相邻子域信息。左图进行行块划分导致更多通信，右图为最优通信情况。阴影格点代表需要进行通信交换的信息

区域分解方法的优势之一是，如果问题规模增长 N 倍，则域边界比域容量增长慢，因 118

此为了解决通信瓶颈，可以增大问题规模。三维区域中选择最优区域分解策略并扩展问题规模或者任务数量的效果将会在 5.3 节中讨论。

在介绍 MPI 接口之后，实现并行 Jacobi 区域分解算法的细节将会在 9.3 节进行讨论。

虽然 Jacobi 算法的收敛性并不十分高效，但是它可以作为更高级算法的基础和原型。此外，它引出了一系列标量优化算法，某些算法在 3.4 节的矩阵转置中会被再次用到（见习题 3.4）。

5.2.2 功能并行性

有时一个庞大的数值问题的求解可以分为或多或少相互独立的子任务，子任务间通过数据交换和同步协同工作，而子任务间在不同的数据集上执行不同代码，这也是任务并行性又称为多程序多数据（Multiple Program Multiple Data，MPMD）的原因。但是每个子任务也可以通过多程序多数据模式在多个处理器上并行执行。

功能并行性在性能方面有利有弊，当不同的子任务有不同的性能特征或者不同的硬件需求时，容易导致性能瓶颈或者负载不均衡等问题。另一方面，任务间重叠执行可以极大提升程序性能。

芯片上处理器核数量不断增长并将核分配给不同任务，下面介绍几种重要的功能并行性的变体，以此来讨论是否进入了功能并行性时代，11.1.2 节在混合编程条件下给出了另一个
119 实例。

1. 实例：主从模式

保留一个计算单元专门进行管理任务，而所有其他计算单元执行实际程序，这种模式称为主从模式（master-worker 模式）。主节点将不同子任务分配给从节点，并收集从节点的结果。一个典型的应用实例为并行光线追踪程序：一个光线追踪器从一个场景的数学表示出发计算现实图像。对于需要渲染的每一个像素，有一束从图像观察者出发的光线照到场景中，击中场景中的物体，并进行反射，依次进行计算，选择彩色的成分。如果所有的计算单元都有场景的一个拷贝，则所有的像素都是独立的，并且可以并行计算。通常由于效率的因素，图像被分为若干工作单元（按行划分或者分块）。当一个从节点计算完一个工作单元后，它向主节点请求一个新的子任务，而主节点维护一个已完成任务队列和一个待完成任务队列。在分布式系统中，一个完成的工作单元将通过网络进行通信。关于主从模式下实现并行光线追踪的详细细节和性能分析参考 [A80，A81]。

主从模式的一个缺点是，当从节点的数量很大时，单一主节点可能会产生通信和性能瓶颈。

2. 实例：功能分解

多物理模拟是功能性分解进行并行化的主要应用。例如行驶中汽车周围的空气流可以利用并行 CFD（计算流体力学，Computational Fluid Dynamics）代码。另一方面，并行有限元模拟可以根据几何和物质特性描述不同的汽车车身结构和流体的交互作用。两种代码需要通过有效的通信层进行交互协同计算。

虽然多物理代码非常流行，但是由于实践中很难在不同功能域之间移动或交互资源，因此通常会导致负载不均衡问题。关于负载不均衡问题的更多信息请见 5.3.9 节。

5.3 并行扩展性

5.3.1 限制并行执行的因素

5.2 节中已经阐述过，可以在不同维度开发并行性。寻找并行性不仅局限于计算领域，而且也在诸如制造、交通流量控制甚至业务处理等领域中发挥作用。可以将所有的执行单元（工人、生产线、等待队列、CPU 等）的运行过程简单看成执行固定时间的工作，这时在理想情况下，利用 N 个工人解决一个顺序执行需要时间 T 的工作需要的时间为 T/N（见图 5-4），这个值称为加速比。 120

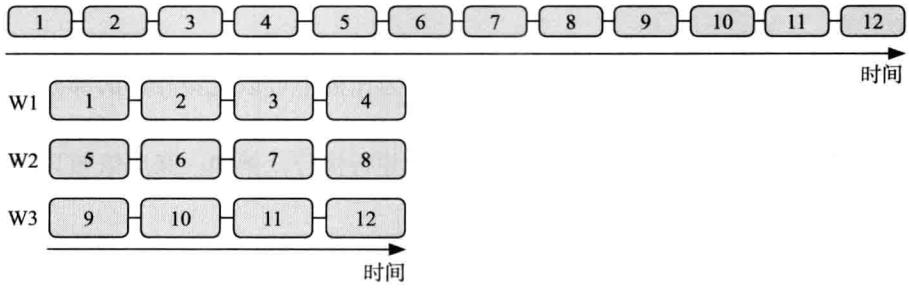


图 5-4 用三个工人（W1、W2、W3）并行化一系列任务（上图）获得完美加速比（左图）

不管选择何种并行化模式，通常不会达到图 5-4 所展示的完美加速比。前面已经介绍过一些原因：并不是所有工作线程都执行相同时间的工作，因为问题并不能（或者不可能）分为同等复杂度的几部分。因此，有时有些线程不得不等待落后线程（见图 5-5），这种负载不均衡由于没有充分利用资源而影响了性能。此外，一些共享资源，例如一些被所有线程共享的工具，会导致并行执行线程的串行化运行（见图 5-6）。最后，并行工作流会要求一些任务间通信，从而导致产生一些并行执行特有的等待开销（见图 5-7）。所有这些因素都限制了并行加速比。任务并行程度通常用一系列可扩展性指标来衡量，如回答下述问题：

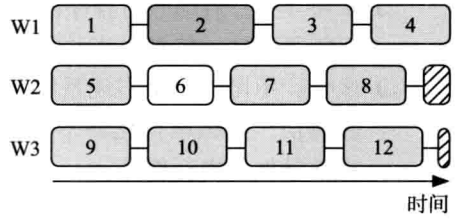


图 5-5 不同线程以不同速度执行不同任务会导致负载不均衡。阴影部分标识未被使用的资源

- 给定 N 个线程，并行解决一个问题最多可以比单个线程时快多少？
- 给定 N 个线程，可以解决比单个线程时大多少的任务规模？
- 通信需求对并行应用程序的性能和可扩展性的影响是什么？
- 有多少资源真正被用到了解决问题的计算中？

接下来的几小节介绍一些重要指标和模型来帮助我们解决上述问题。 121

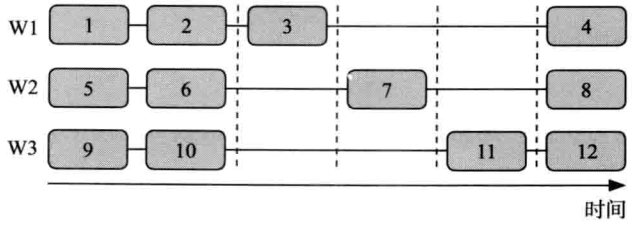


图 5-6 并行执行的瓶颈。任务 3、7 和 11 不能跨过分割线重叠执行

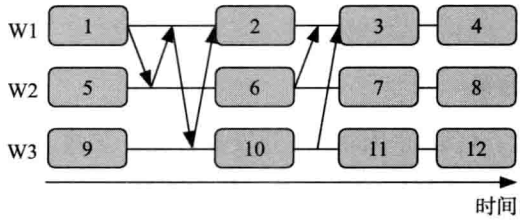


图 5-7 如果无法与其他计算重叠，通信进程（箭头表示消息）就限制了可扩展性

5.3.2 可扩展性指标

为了定义可扩展性指标，我们首先定义一些需要的测量指标。一般设所有任务规模（工作量）为 $s+p=1$，s 为任务中不能并行的部分而 p 为任务中可以完美并行的部分。任务中存在不能并行部分的原因包括：

- 算法限制：有些操作由于某些原因不能被并行执行，例如：互斥依赖只能在线程间顺序执行，必须以某个特定排序执行。
- 瓶颈：计算机系统中存在大量共享资源，例如核中的执行单元，多核芯片内存中的共享数据通路，I/O 设备。对共享资源的并行访问会导致串行化执行，即使算法可以完全并行执行，由于共享资源的限制也可能会导致瓶颈。
- 启动开销：启动并行程序的开销。即使大规模并行系统做了深入优化，启动开销也是不能避免的。特别是当并行程序执行时间较短时，启动开销对性能就有更大的影响。
- 通信：不可能完全实现系统各子部分间的完全并发通信，见 4.5 节。如果并行程序需要通信，那么串行化将不可避免。5.3.6 节将把通信融入可扩展性指标中，而不只是将常量加入串行部分。

122

首先我们假定应用 N 个线程解决固定规模的问题。正则化单线程（串行）时间

$$T_{\mathrm{f}}^{\mathrm{s}} = s+p \tag{5-1}$$

为 1。

应用 N 个线程解决需要的时间为：

$$T_{\mathrm{f}}^{\mathrm{p}} = s+\frac{p}{N} \tag{5-2}$$

由于无论有多少线程执行任务，串行和并行执行的工作量都是固定的，所以这个指标称为强可扩展，此时并行化的目标为最小化给定问题并行执行的时间。

如果执行时间不是主要关注目标，而是由于内存限制，希望在并行执行时处理更大规模的问题，就需要按 N 的一个指数规模扩展问题大小，即 $s+pN^{\alpha}$，此处 α 大于零或者没有参数，并且假设问题的串行部分 s 固定。定义可扩展问题（大小可变）串行执行时间为

$$T_{\mathrm{v}}^{\mathrm{s}} = s+pN^{\alpha} \tag{5-3}$$

同样，得到并行执行时间为

$$T_{\mathrm{v}}^{\mathrm{p}} = s+pN^{\alpha-1} \tag{5-4}$$

这个指标称为弱可扩展。通常令 $\alpha=1$。其他 N 的函数可以用来计算弱可扩展性，但是本书使用 N^{α}。

我们将会看到不同的并行性指标强调不同的“性能”意义会导致相反的结果。

5.3.3 简单可扩展性定律

程序加速比可以定义为固定规模问题的串行时间与并行时间之比。除非特别声明，下面定义性能为工作时间。固定规模问题 $s+p$ 的串行性能为：

$$P_{\mathrm{f}}^{\mathrm{s}}=\frac{s+p}{T_{\mathrm{f}}^{\mathrm{s}}}=1 \tag{5-5}$$

并行性能为：

$$P_{\mathrm{f}}^{\mathrm{p}}=\frac{s+p}{T_{\mathrm{f}}^{\mathrm{p}}(N)}=\frac{1}{s+\dfrac{1-s}{N}} \tag{5-6}$$

123

程序加速比为：

$$S_{\mathrm{f}}=\frac{P_{\mathrm{f}}^{\mathrm{p}}}{P_{\mathrm{f}}^{\mathrm{s}}}=\frac{1}{s+\dfrac{1-s}{N}}\text{，Amdahl 定律} \tag{5-7}$$

这样就导出了 1967 年 [M45] 由 Gene Amdahl 首先导出的 Amdahl 定律。它限制了当 N 趋于无穷大时程序的加速比为 $1/s$。这个定律回答了“当利用 N 个 CPU 并行处理一个问题时所能获得的最快速度（以运行时间的方式）”，问题的答案随问题工作量定义的不同而不同。如果定义的工作量仅为计算中可并行化部分（必须合理地将其定义为并行部分），则对于固定规模任务结果将会非常不同。串行性能为：

$$P_{\mathrm{f}}^{sp}=\frac{p}{T_{\mathrm{f}}^{\mathrm{s}}}=p \tag{5-8}$$

并行性能为：

$$P_{\mathrm{f}}^{pp}=\frac{p}{T_{\mathrm{f}}^{\mathrm{p}}(N)}=\frac{1-s}{s+\dfrac{1-s}{N}} \tag{5-9}$$

应用程序加速比为：

$$S_{\mathrm{f}}^{p}=\frac{P_{\mathrm{f}}^{pp}}{P_{\mathrm{f}}^{sp}}=\frac{1}{s+\dfrac{1-s}{N}} \tag{5-10}$$

又导出了 Amdahl 定律。这时 P_{f}^{pp} 和 $S_{\mathrm{f}}^{p}(N)$ 不再相等。虽然此时工作的定义不同，但可扩展性并没有改变。不过性能有一个因子为 p 的差异。

在弱可扩展情况中，工作量随着 CPU 数目增长，此时需要回答“给定 N 个 CPU 可以解决比单个线程时大多少的任务规模”，当 N=1 时串行性能为：

$$P_{\mathrm{v}}^{\mathrm{s}}=\frac{s+p}{T_{\mathrm{f}}^{\mathrm{s}}}=1 \tag{5-11}$$

基于式（5-3）和式（5-4），并行性能为：

$$P_{\mathrm{v}}^{\mathrm{p}}=\frac{s+pN^{\alpha}}{T_{\mathrm{v}}^{\mathrm{p}}(N)}=\frac{s+(1-s)N^{\alpha}}{s+(1-s)N^{\alpha-1}}=S_{\mathrm{v}} \tag{5-12}$$

这样又与程序加速比一致。当 α=0 时（强可扩展）我们又得到了 Amdahl 定律。当 $0<\alpha<1$ 时，对于较大数目的处理器，有

$$S_{\mathrm{v}}\xrightarrow{N\gg 1}\frac{s+(1-s)N^{\alpha}}{s}=1+\frac{p}{s}N^{\alpha} \tag{5-13}$$

124 这与 N^{α} 呈线性关系，这样，弱可扩展性容许我们突破 Amdahl 定律的限制，即使对较小的 α，也能得到更高的没有上限的性能。理想情况为 α=1，式（5-12）简化为

$$S_{\mathrm{v}}(\alpha=1)=s+(1-s)N,\ \text{Gustafson 定律} \tag{5-14}$$

即使对于较小的 N，加速比也与 N 呈线性关系，这称为 Gustafson 定律 [M46]。注意加速比公式前半部分依赖一个串行比例 s，这样随着 N 的增加，s 的比重变得非常小。

与前面的讨论类似，现在我们将注意力转移到对工作量的定义上来，首先假定工作量只包括并行执行部分 p，这样串行性能为：

$$P_{\mathrm{v}}^{sp}=p \tag{5-15}$$

并行性能为：

$$P_{\mathrm{v}}^{pp}=\frac{pN^{\alpha}}{T_{\mathrm{v}}^{\mathrm{p}}(N)}=\frac{(1-s)N^{\alpha}}{\mathrm{s}+(1-s)N^{\alpha-1}} \tag{5-16}$$

程序加速比为：

$$S_{\mathrm{v}}^{p}=\frac{P_{\mathrm{v}}^{pp}}{P_{\mathrm{v}}^{sp}}=\frac{N^{\alpha}}{s+(1-s)N^{\alpha-1}} \tag{5-17}$$

同样，加速比和性能又有一个因子为 p 的区别。相比式（5-14）而言，α=1 时程序加速比与 N 呈线性关系。所以式（5-17）说明程序可以被完全扩展。

5.3.4 并行效率

在考虑扩展性的同时，另一个关心的问题是给定资源的使用效率如何，即 CPU 的计算能力有多少被真正用在了并行任务的执行过程中（接下来我们假设当一个线程执行程序串行部分时，其他线程均等待）。通常并行效率定义如下：

$$\varepsilon=\frac{N\text{ 个 CPU 上的性能}}{N\times 1\text{ 个 CPU 上的性能}}=\frac{\text{加速比}}{N} \tag{5-18}$$

因为 α 趋于 0 时会退化到 Amdahl 定律，我们只考虑弱可扩展。当任务工作量定义为 $s+pN^{\alpha}$ 时，我们有：

125
$$\varepsilon=\frac{S_{\mathrm{v}}}{N}=\frac{sN^{-\alpha}+(1-s)}{sN^{1-\alpha}+(1-s)} \tag{5-19}$$

当 α=0 时公式变为 $1/(sN+(1-s))$，这也是 Amdahl 定律的结果，当 N 增大时趋于 0。当 α=1 时我们得到 $s/N+(1-s)$，这是由于随着 CPU 数量 N 的增加，更多的 CPU 时钟周期被浪费，从串行 N=1 时的 $\varepsilon=s+p$=1，到 N 较大时的 $1-s=p$。甚至 N 增大时，弱可扩展也使得我们至少利用一部分 CPU 资源，但是浪费的 CPU 资源随着 N 而线性增长。

当我们将工作量定义为 pN^{α} 时，有不同的结果：

$$\varepsilon_{p}=\frac{S_{\mathrm{v}}^{p}}{N}=\frac{N^{\alpha-1}}{s+(1-s)N^{\alpha-1}} \tag{5-20}$$

当 α=1 时得到 ε_p=1，这意味着所有的资源都被充分利用。虽然 s 较大时大量 CPU 资源未被利用，但是在弱可扩展性的意义下，我们仍错误地认为没有浪费时钟周期。我们举一个例子：假设某个程序在并行执行部分进行浮点数运算，并且只占 10% 的串行计算时间，利用 α=1 时的弱可扩展分析，可以得到 MFlop/s 性能与 CPU 数量的关系（如图 5-8 所示），虽然使用 N 个 CPU 时大部分处理器 90% 的时间都在等待，但是图中显示 MFlop/s 性能却随着 N 的增加而增加，这种方式的分析会导致错误。

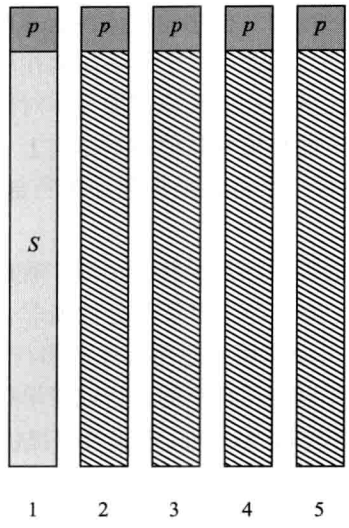


图 5-8 当任务工作量只包括程序可并行部分时的弱可扩展。虽然时间随着 CPU 的增加完全可扩展，即 ε_p=1，但当 $s \gg p$ 时有大量资源（阴影部分）未被使用

5.3.5 串行性能与强可扩展性

为了检查某个性能模型是否适用于一个程序，需要测试处理器可扩展性并通过最小二乘法确定模型中的参数，图 5-9 展示了程序在两种不同并行体系结构中的强可扩展性实例，所有性能值都归一化到第一个体系结构上的单处理器性能，通过最小二乘法确定程序串行部分 s 以满足 Amdahl 定律（式（5-7））。 126

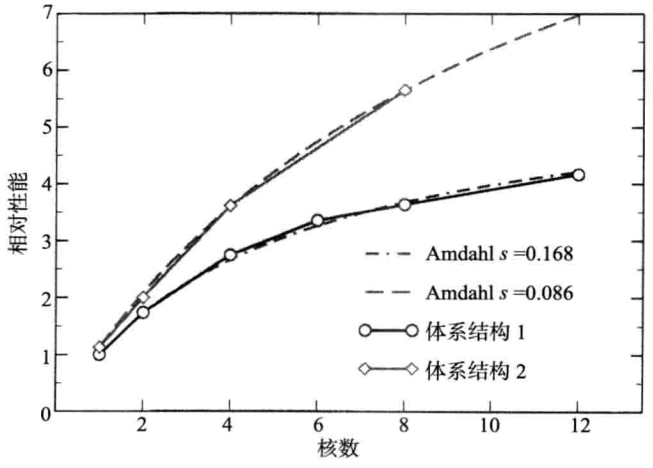


图 5-9 在两个不同体系结构上基准测试程序的性能与处理器数目的实测数值（强可扩展）。虽然单处理器情况下两个体系结构的性能十分接近，但是由于程序串行部分 s 在第二个体系结构上的数值较小，因此有更好的强可扩展性

可以看到在单核情况下两种体系结构的性能差别不大，但是第二个体系结构却有良好的强可扩展性，这种现象的原因在于，程序可并行部分为计算受限，然而程序串行部分为访存受限，虽然两个体系结构的单核处理器性能峰值相同，但是第二个处理器体系结构有更宽

的内存通路。随着线程数量的增加，程序的整体性能不再依赖于计算速度，而程序访存受限的顺序执行部分变得更为重要。因此第二个体系结构在可扩展性方面表现较好。这个例子表明不仅要考虑非并行部分的比例，还要考虑这段程序对体系结构的具体需求，并且一个体系结构上的可扩展性并不能一定能移植到其他体系结构上。为了提高程序强可扩展性的体系结构可能会采取异构架构，包含一个可以处理程序串行部分的硬件，这也适用于众核处理器[R39,M47,M48]。

从优化角度看，强可扩展性的增加最终会受限于程序中的串行执行部分（这是经济学中报酬递减法则的一种变体）。虽然程序串行部分性能非常难以改变，但是如果第2章和第3章提到的标准标量优化方法可以应用到应用程序的串行执行部分，就可以改进强可扩展性。需要回答的问题是，是否需要对应用程序的并行或者串行部分进行标量优化。注意与可扩展
127 性不同，性能是一个相对的指标，Amdhal 定律可以提供一个粗略的指导。假设程序串行部分的加速比为 $\xi>1$，并行性能（见式（5-6））为：

$$P_{\mathrm{f}}^{s,\xi}=\frac{1}{\frac{s}{\xi}+\frac{1-s}{N}} \tag{5-21}$$

另一方面，如果仅优化并行执行部分（以相同因子优化），则有：

$$P_{\mathrm{f}}^{p,\xi}=\frac{1}{s+\frac{1-s}{\xi N}} \tag{5-22}$$

上述两个公式之比决定交点，即优化程序串行部分花费的线程数：

$$\frac{P_{\mathrm{f}}^{s,\xi}}{P_{\mathrm{f}}^{p,\xi}}=\frac{\xi s+\frac{1-s}{N}}{s+\xi\frac{1-s}{N}}\geqslant 1 \Longrightarrow N\geqslant\frac{1}{s}-1 \tag{5-23}$$

可以看到结果不依赖于 ξ，并且为加速比达到 Amdahl 定律预测的最大之值的一半时的线程数目。如果 $s\ll 1$，并行效率 $\varepsilon=(1-s)^{-1}/2$ 接近 0.5，并且及即使增大 N 也不能再有大幅提高。这样就需要首先优化程序并行部分，除非程序代码的并行效率非常差（可能的主要原因是由于缺少内存才对程序进行并行化）。

然而，需要注意实际上对串行和并行部分代码而言可能无法获得 ξ 倍加速比，所以交点也会相应地移动。在上例中（参见图 5-9），并行部分的主要开销是性能接近峰值的矩阵乘算法，因此在给定 N 时，加速串行部分是改进性能的唯一选择。

5.3.6 改进的性能模型

由于底层模型可能不再隐藏一些例如通信、负载不均衡和并行启动开销之类的组件，因此简单的模型例如 Amdahl 定律和 Gustafson 定律就不再适用。一个简单的改进模型的例子是包含通信开销，简单起见我们假定通信和计算不能重叠执行（见图 5-7），这个假设对于大多数并行体系结构均成立。在计算并行性能时，必须要加入通信时间作为并行运行时的正确术语：

$$T_{\mathrm{V}}^{\mathrm{pc}}=s+pN^{\alpha-1}+c_{\alpha}(N) \tag{5-24}$$

由于通信是并行化导致的开销，因此通信开销 $c_{\alpha}(N)$ 不属于工作量，因此不能算有效工作量，并行加速比为

$$S_{\mathrm{v}}^{\mathrm{c}}=\frac{s+pN^{\alpha}}{T_{\mathrm{v}}^{\mathrm{pc}}(N)}=\frac{s+(1-s)N^{\alpha}}{s+(1-s)N^{\alpha-1}+c_{\alpha}(N)} \tag{5-25}$$

128

有一些 $c_\alpha(N)$ 所依赖的参数，可能是一些简单的函数，也可能是一些没有解析式的函数。此外，我们假定所有线程的通信量都是一样的。在处理器和内存通信中，传递一个消息的时间是启动通信时延迟时间之和 λ 和流传输部分时间 $\kappa=n/B$，其中 n 为消息长度而 B 为带宽（4.5.1 节有一个实际例子）。一些特殊情况如下：

- $\alpha=0$，阻塞网络：如果通信网络具有总线特征（见 4.5.2 节），即一个时刻只允许一个消息在网络中传递，通信的开销与处理器数目 N 无关，此时 $c_\alpha(N)=(\kappa+\lambda)N$，得到：

$$S_{\mathrm{v}}^{\mathrm{c}}=\frac{1}{s+\dfrac{1-s}{N}+(\kappa+\lambda)N}\xrightarrow{N\gg 1}\frac{1}{(\kappa+\lambda)N} \tag{5-26}$$

可以看到程序性能受限于通信性能，甚至对于较大规模的处理器数目，加速比趋于 0，这是一种常见的模式，例如某些共享资源例如内存通路、I/O 设备甚至片上计算单元的共享访问。

- $\alpha=0$，非阻塞网络，固定通信开销：如果通信网络可以容纳 $N/2$ 个消息同时传递并没有引起冲突（见 4.5.3 节），并且消息大小与处理器数目 N 无关，此时 $c_\alpha(N)=\kappa+\lambda$，得到

$$S_{\mathrm{v}}^{\mathrm{c}}=\frac{1}{s+\dfrac{1-s}{N}+\kappa+\lambda}\xrightarrow{N\gg 1}\frac{1}{s+\kappa+\lambda} \tag{5-27}$$

这种情况非常类似于 Amdahl 定律的情形，程序会在一个比没有通信时更低的数值达到最高加速比。

- $\alpha=0$，非阻塞网络，具有 ghost 层通信的区域分解方式：在这种情况下，强可扩展中的通信开销随着处理器数目 N 的增加而减少，$c_\alpha(N)=\kappa N^{-\beta}+\lambda$，对于 $\beta>0$，当处理器数目 N 增大时程序性能将受限于程序串行部分 s 和通信延迟，得到：

$$S_{\mathrm{v}}^{\mathrm{c}}=\frac{1}{s+\dfrac{1-s}{N}+\kappa N^{-\beta}+\lambda}\xrightarrow{N\gg 1}\frac{1}{s+\lambda} \tag{5-28}$$

对于 5.2.1 节的例子而言，三维情况下的区域分解有 $\beta=2/3$。

- $\alpha=1$，非阻塞网络，具有 ghost 层通信的区域分解方式：当程序规模与处理器数目 N 线性同步增长时，最终每个处理器会达到一个与 N 无关的数值。在弱可扩展情况下，将会达到线性可扩展性，只不过有一个总体的性能降低因子：

$$S_{\mathrm{v}}^{\mathrm{c}}=\frac{s+pN}{s+p+\kappa+\lambda}\xrightarrow{N\gg 1}\frac{s+(1-s)N}{1+\kappa+\lambda} \tag{5-29}$$

129

图 5-10 显示了四个不同情况，$s=0.05$，$\kappa=0.005$，$\lambda=0.001$，并且与 Amdahl 定律进行比较。注意现在所使用的模型与实际应用程序的情况有较大区别。考虑这样一个例子，一个应用程序的数据比单一处理器的 cache 大小要大，但是小于所有 N_c 处理器的 cache 容量之和。性能受限因素例如程序串行部分、通信部分等可以忽略不计，或者是超量补偿，此时有 $S_{\mathrm{v}}^{\mathrm{c}}(N)>N$，对于某一范围内的 N 这种情况成为超线性加速比，只在程序规模增长率低于 N 的情况下出现，亦即 $\alpha<1$。参考 6.2 节和习题 7.2。

注意这些模型只有在 $N>1$ 的情况下成立，由于串行执行时没有通信开销，因此利用曲线拟合过程来确定某些代码中的参数时要忽略 $N=1$ 的情况。

当在并行机上执行并行程序时通常要确定最优的处理器数目 N。从用户的角度看，处理器数目 N 越大越好，这样才能降低问题求解时间。但是这会极大浪费资源，性能达到最大时并行效率会很低。习题 5.2 给出了一个解决该问题的性能模型。注意如果增加内存是并行化的主要原因，那么较低的并行效率是可以接受的。

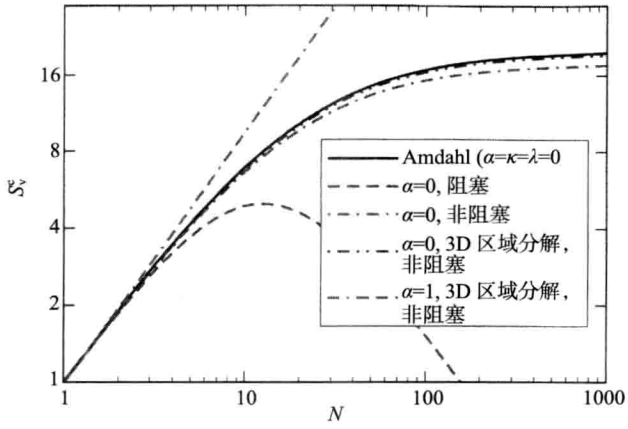


图 5-10 $s=0.05$ 时不同模型下预测的并行可扩展性。除了 Amdahl 情况下，其他均设 $\kappa=0.005$，$\lambda=0.001$

5.3.7 选择正确的扩展性基准

当代的高性能计算机的并行性规模非常大，前面几节我们讨论了并行计算机不同的构建方法：可以是共享内存多个多核处理器节点，并在不同层次上由通信网络互联。因此并行系统总是包含多级层次化构件。将并行代码从单个 CPU 扩展到多个 CPU 时需要考虑这种层次化结构，否则将会得到错误的结论。

130 图 5-11 展示了程序在一个 4 路处理节点上的强可扩展性，假设程序遵循式（5-26）给出的通信模型，则应用最小二乘法求解程序串行部分 s 和单个处理器通信时间 $k=\kappa+\lambda$（主板）。由于在 16 个核的情况下加速比约为 4，所以设定 $s=0.2$，通信开销可以忽略不计。但是这个估计比较粗糙，特别是处理器数目较小时。这样我们得到的结论是节点内的可扩展性的首要因素与程序串行部分和通信不同，并且式（5-26）并不是对所有核都有效。图 5-11 中的右半部分显示了归一化到单节点四核处理器性能的可扩展性结果，即选择一个不同的可扩展性基准。此时式（5-26）的模型完全适合于此情况，并且产生了不同的拟合参数，此时通信参数是一个重要的影响因子（$s=0.01$，$k=0.05$）。图 5-11 中的左侧显示了一个节点内的可扩展性数值，曲线是一个典型的内存受限型曲线。在一个类似图 4-4 所示的节点体系结构上，使用双路双核可能会导致带宽瓶颈，这可以从单核到双核的非常有限的加速比中推断出来，但是使用另一个处理器的核心会极大提升性能。

总之，并行体系结构上的可扩展性要依赖于所选取的可扩展基准。在典型的集群系统中，共享内存的多处理器节点通过高速网络互连，这意味着节点内和节点间的可扩展性不同。这个原理也适用于其他具有层次结构的系统例如多路多核共享内存系统（见 4.2 节），甚

至在现代多核处理器中复杂的 cache 组和线程结构（见 1.4 节）。

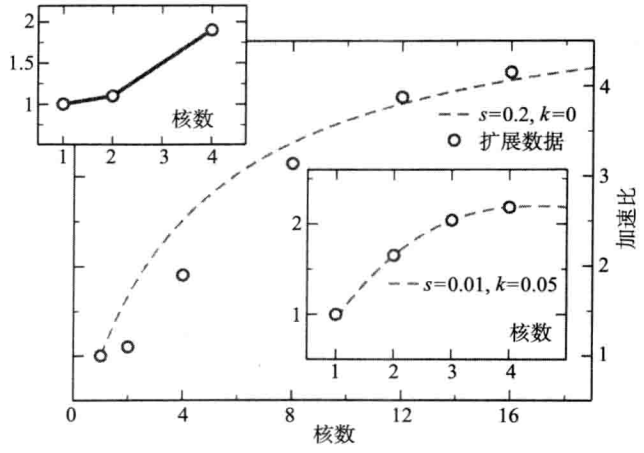


图 5-11　四路节点层次化系统中的加速比与 CPU 数目，如果选择不同的比较基准，可扩展性会有不同的结果。右侧子图表示选择节点性能作为基准的可扩展性。左侧子图表示选择 CPU 性能作为基准的可扩展性

5.3.8　案例分析：低速处理器计算机能否变得更快

如果并行系统其他部分都不变而只更换一个更慢的 CPU（或者使用一个未优化的单处理器节点），那么由于通信相对于计算会降低开销比率，所以会改进应用程序的可扩展性。因此一个“良好可扩展性”计算机系统建立在低速 CPU 和良好的通信网络基础之上。为了检验这个推断，我们建立了一个低速计算机系统性能模型，这里低速意味着串行执行时间 μ>1 而不是 1，即 CPU 的处理速度为 μ^{-1}。图 5-12 展示了低速计算机如何工作，如果程序在 μN 个低速处理器而不是 N 个快速处理器上执行并且每个 CPU 的通信开销降低，就依然有可能降低整体运行时间。但是这样构建一个并行系统是否真正有效，在此基础上建立一个并行计算机系统是否可行还需要回答下列问题：

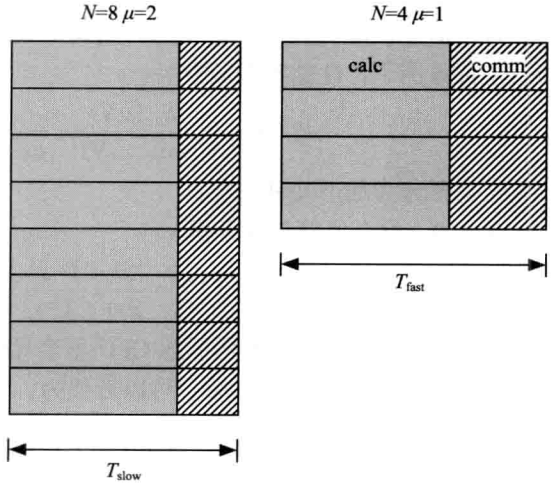


图 5-12　如果 CPU 通信开销随 N 增大而降低，那么用 $2N$ 个低速 CPU(左图）而不是 N 个快速 CPU(右图）会提升性能。μ=2 是低速 CPU 和高速 CPU 的性能比

- 使用 μN 个低速处理器而不是标准的 N 个处理器是否会提高整体性能？
- 为了依赖低速处理器获得更高性能，通信开销需要满足何种条件？
- 这种情况对于强可扩展性和弱可扩展性是否都成立？
- 期望的性能提升是多少？
- 在同样的功耗情况下，低速处理器是否比标准处理器的机器做更多的工作？

对于最后一个问题，如果不考虑通信开销，我们已经在1.4节讨论过类似情形（多核传
132 输顺序）。额外的低效通信性能可能会极大地影响最终结果。更重要的是，CPU核只占整体并行系统功耗开销的一部分，只包括CPU的功耗或性能模型是不完整的。因此我们考虑其他问题，从图5-12可以看到一个理想的性能模型需要包括通信组件。

1. 强可扩展

假设问题规模固定，利用式（5-24）包括的通用通信模型，低速计算机的加速比为：

$$S_\mu(N)=\frac{1}{s+(1-s)/N+c(N)/\mu} \tag{5-30}$$

对于$\mu>1$和$N>1$，只要$c(N)\neq 0$，该式就大于$S_{\mu=1}(N)$，这样只要考虑通信开销，低速处理器就有更好的可扩展性。

由于在相同加速比μ^{-1}的CPU上可扩展性与串行和并行性能之比相关，所以可扩展性本身并不是程序性能的理想度量，我们需要比较μN个低速CPU与N个标准处理器之间的性能绝对比值：

$$A_\mu^s(N):=\frac{S_\mu(\mu N)}{\mu S_{\mu=1}(N)}=\frac{s+(1-s)/N+c(N)}{\mu s+(1-s)/N+c(\mu N)} \tag{5-31}$$

如果$\mu>1$，上式在如下条件下将大于1，如果：

$$c(\mu N)-c(N)<-s(\mu-1) \tag{5-32}$$

因此，如果假定该条件对所有μ都成立，$c(N)$必须是N的递减函数。当$s=0$时只要斜率为负即可，即通信开销随着N的增加而降低，这也是图5-12的观察结果。

为了估计性能提升大小，我们利用式（5-28）在非阻塞网络中考虑笛卡儿坐标区域分解的一个特殊例子，性能提升函数为：

$$A_\mu^s(N):=\frac{S_\mu(\mu N)}{\mu S_{\mu=1}(N)}=\frac{s+(1-s)/N+\lambda+\kappa N^{-\beta}}{\mu s+(1-s)/N+\lambda+\kappa(N\mu)^{-\beta}} \tag{5-33}$$

这又区分为如下几种情况：

- $\kappa=0$：没有通信带宽开销

$$A_\mu^s(N)=\frac{s+(1-s)/N+\lambda}{\mu s+(1-s)/N+\lambda}\xrightarrow{N\to\infty}\frac{s+\lambda}{\mu s+\lambda} \tag{5-34}$$

133 该式总小于1，这样使用低速CPU不会导致性能提升，并且功耗的提升也十分有限。

- $\kappa\neq 0$，$\lambda=0$，式（5-33）可以近似为：

$$A_\mu^s(N)=\frac{1}{\mu s}(s+\kappa N^{-\beta}(1-\mu^{-\beta}))+\mathcal{O}(N^{-2\beta})\xrightarrow{N\to\infty}\frac{1}{\mu} \tag{5-35}$$

很明显，当$s\neq 0$，$\kappa\neq 0$时导致了相反的效果，即对较大的N，程序串行部分是性能的主要因素，并且$A_\mu(N)<1$，而对较小的N，如果s较小将会得到$A_\mu^s(N)>1$。

- $s=0$：在强可扩展性条件下这个假设是不现实的，但是这给出了低速CPU能获得最大性能的上线，通信带宽开销随着N的增大而降低，因此减少了与κ相关的因子的系数，特别是当通信延迟较低时：

$$A_\mu^s(N)=\frac{N^{-1}+\lambda+\kappa N^{-\beta}}{N^{-1}+\lambda+\kappa(N\mu)^{-\beta}}\xrightarrow{N\to\infty,\lambda>0}1^+ \tag{5-36}$$

一般情况下$\kappa\neq 0$，$\lambda\neq 0$并且有$0<\beta<1$，这是该函数随着N趋于无穷大而无限接近1，并且有最大值$N_{MA}=(1-\beta)/\beta\lambda$，因此最大的性能提升为：

$$A_{\mu}^{s,\max}=A_{\mu}^{s}(N_{MA})=\frac{1+\kappa\beta^{\beta}X^{\beta-1}}{1+\kappa\beta^{\beta}X^{\beta-1}\mu^{-\beta}},\ \text{其中}\ X=\frac{\lambda}{1-\beta} \tag{5-37}$$

该式随着 λ 趋近于 0 而接近 μ^{β}。典型的可扩展高性能计算机系统通常配备的低速 CPU 有 $2\lesssim\mu\lesssim4$，因此对 5.2.1 节的例子而言，最优三维区域分解算法（β=2/3）理论上最大的性能提升为 $1.5\lesssim A^{s,\max}\lesssim2.5$。

需要强调的是 s=0 的假设在强可扩展性分析下并不十分正确，因为就像对较大的 N 而言网络通信延迟会占主要开销一样，程序串行部分随着 N 的增加也会占据主要地位。因此在强可扩展条件下使用低速处理器的应用范围比较有限。

即使在技术上我们可以设置 μ 为较大数值并获得更大性能提升，但是应用程序要具有足够的并行性以利用大规模处理器，这不仅涉及只关心程序串行部分 s 的 Amdahl 定律，该定律预测在处理器规模增大时程序串行部分将成为性能瓶颈，并且还受限于程序并行粒度（流体网格数目或者例子数量等），这也限制了可以利用的线程数目。

2. 严格弱可扩展

应用 Gustafson 可扩展性（工作量与 N 成正比）与通用通信模型，低速计算机的加速比为：

$$S_{\mu}(N)=\frac{[s+(1-s)N]/(\mu+c(N))}{\mu^{-1}}=\frac{s+(1-s)N}{1+c(N)/\mu} \tag{5-38}$$

134

得到效益函数如下：

$$A_{\mu}^{w}(N):=\frac{S\mu(\mu N)}{\mu S_{\mu=1}(N)}=\frac{[s+(1-s)\mu N][1+c(N)]}{[s+(1-s)N][\mu+c(\mu N)]} \tag{5-39}$$

忽略程序串行部分 s，如果 $c(N)>c(\mu N)/\mu$，该式大于 1，即通信函数开销可以随着处理器数目 N 的增长而提高，但是增长率要小于线性增长。

依然选取笛卡儿坐标区域分解算法来进行定量分析，弱可扩展性增加了通信开销，并且通信开销独立于处理器数目 N，见式（5-29）。设 $\tilde{\lambda}:=\kappa+\lambda$，低速计算机的加速比为：

$$S\mu(N)=\frac{s+(1-s)N}{1+\tilde{\lambda}\mu^{-1}} \tag{5-40}$$

因此性能效益函数如下：

$$A_{\mu}^{w}(N):=\frac{S\mu(\mu N)}{\mu S_{\mu=1}(N)}=\frac{(1+\tilde{\lambda})[s+(1-s)\mu N]}{[(1-s)N+s](\tilde{\lambda}+\mu)} \tag{5-41}$$

考虑以下几种特殊情况：

- $\tilde{\lambda}$ =0：不考虑通信开销

$$A_{\mu}^{w}(N)=\frac{(1-s)N+s/\mu}{(1-s)N+s}=1-\frac{\mu-1}{\mu N}s+\mathcal{O}(s^{2}) \tag{5-42}$$

非常类似于强可扩展情况式（5-34），该式小于 1。

- $s=0$：如果程序具有完全并行性，并且 $\mu>1$，那么性能效益函数大于 1，并且独立于处理器数目 N：

$$A_{\mu}^{w}(N)=\frac{1+\tilde{\lambda}}{1+\tilde{\lambda}/\mu}=\begin{cases}\underrightarrow{\mu\gg\tilde{\lambda}}\ 1+\tilde{\lambda}\\ \underrightarrow{\tilde{\lambda}\gg1}\ \mu\end{cases} \tag{5-43}$$

然而对于较小的 $\tilde{\lambda}$ 没有性能提升。即使对 $\tilde{\lambda}$ =1，即通信开销等于串行执行时的开销，那么当 μ 取典型值 $2\lesssim\mu\lesssim4$ 时，仅有 $1.33\lesssim A^{w}\lesssim1.6$。

上面我们假定选用 μ 倍数目的低速处理器处理 μ 倍规模的同样问题，这称为严格弱可扩展，在不同的问题规模上比较快速和低速处理器，但是这并不具有很强的现实意义，特别是实际的运行时间也相应增长，从程序性能效益函数 A_{μ}^{w} 的角度看，即使其数值大于 1，也忽视了更重要的解决时间数值。如果 $A_{\mu}^{w} \lesssim \mu$ 则该问题可以被补偿，但是从式（5-43）得知该假
135 设不可能成立。

3. 改进的弱可扩展

实际中应该将问题规模增长为 N 倍（标准处理器数目），而不是 μN 倍，这样低速 CPU 的内存配置相应地可以小 μ 倍，这才是高性能计算系统中常用的可扩展概念，这个性能模型包括了弱可扩展和强可扩展。

必须区分线程数量和问题规模情况下的收益函数，因此初始的加速比函数为：

$$S_{\mu}^{\mathrm{mod}}(N,W)=\frac{[s+(1-s)W]/[\mu s+\mu(1-s)W/N+c(N/W)]}{[s+(1-s)]/\mu}=\frac{s+(1-s)W}{s+(1-s)W/N+c(N/W)\mu^{-1}} \tag{5-44}$$

这里 N 是线程数量，W 代表并行程序问题规模。如果 $W=N$ 并且 $c(1)=\tilde{\lambda}$ 则该式变为严格弱可扩展性。式中 $c(N/W)$ 一项表示强可扩展组件，当 $N>W$ 时能有效降低通信开销，这样就得到改进的弱可扩展性收益函数如下：

$$A_{\mu}^{\mathrm{mod}}(N):=\frac{S_{\mu}^{\mathrm{mod}}(\mu N,N)}{\mu S_{\mu=1}^{\mathrm{mod}}(N,N)}=\frac{1+c(1)}{1+s(\mu-1)+c(\mu)} \tag{5-45}$$

由于当线程数目从 N 增长到 μN 时我们保持问题规模不变，所以该式特别之处在于其值与处理器数目 N 无关，当 $N=1$ 时（见式（5-32））该式的性能提升值与式（5-32）强可扩展性相同：

$$c(\mu)-c(1)<-s(\mu-1) \tag{5-46}$$

考虑笛卡儿坐标区域分解算法，得到 $c(\mu)=\lambda+\kappa\mu^{-\beta}$，因此有

$$A_{\mu}^{\mathrm{mod}}(N)=\frac{1+\lambda+\kappa}{1+s(\mu-1)+\lambda+\kappa\mu^{-\beta}} \tag{5-47}$$

当 $s=0$ 并且根据 κ 和 λ 中的顺序有：

$$A_{\mu}^{\mathrm{mod}}(N)=1+(1-\mu^{-\beta})\kappa-(1+\mu^{-\beta})\lambda\kappa+\mathcal{O}(\lambda^{2},\kappa^{2}) \tag{5-48}$$

可以看到通信带宽开销 κ 是该式的主项，所以与式（5-43）所展示的严格弱可扩展性不同，延迟在这里不占主要地位。即使对 $\kappa=1$，$\lambda=0$，$\beta=2/3$ 并且 $2\lesssim\mu\lesssim4$ 时，得到 $1.2\lesssim A^{\mathrm{mod}}\lesssim1.4$。一般而言，如果具有较大的带宽开销和较低的延迟，改进的弱可扩展是衡量拥的低速处理器的并行计算机系统的一个比较合理的模型，它对 μ 的依赖不大，并且当 μ 趋于无穷大时性能提升趋于 $1+\kappa$。

136 总之，我们从理论上分析了使用低速处理器构建大规模并行系统的可能性，再结合功耗开销和应用程序性能的降低，就可以在一定程度上解决“功耗 - 性能困境”，这也是 IBM Blue Gene 超级计算机系统的实际解决方案 [V114,V115]。但是要注意并不是所有的程序都适用于大规模并行系统，并且在达到良好可扩展性的各个方面都需要平衡（例如构建完全非阻塞胖树网络通信系统成本非常昂贵）。

5.3.9 负载不均衡

没有经验的 HPC 程序员在试图寻找并行程序扩展性差的原因时，精力往往集中在所

使用平台的硬件细节和所使用的并行方法的缺点与不足（如通信开销、同步丢失、伪共享、NUMA 本地化以及带宽瓶颈等）上。当所有这些缺陷成为扩展性差的可能原因时（本书的其他章节会详细讨论），负载不均衡往往被忽略。当有些工作线程比其他线程更早达到同步点时（见图 5-5），负载不均衡发生，此时就会导致至少一个工作线程处于空闲状态而其他线程依旧在执行有用任务，从而导致资源的低效利用。

如果没有对任务分配情况的进一步假设，很难通过一个简单模型来表征负载不均衡所导致的结果。对性能的影响也同样不容易判断：如图 5-13 所示，当有些工作线程到达同步点的时间较晚（“lagger”）时，会导致其他大多数工作线程（即主要工作线程）空闲等待一段时间，从而导致明显的性能损失。但另一方面，一些极少数“speeder”即完成任务较早的线程，可能对性能影响不大，因为累积的线程等待时间可以忽略（参见图 5-14）。

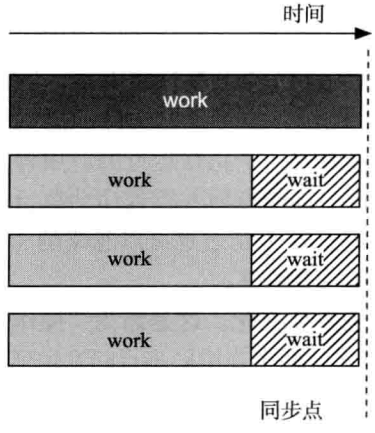


图 5-13 少数工作线程为“lagger”的负载不均衡现象（这里只有一个）。大量的资源没有得到充分利用（阴影线区域）

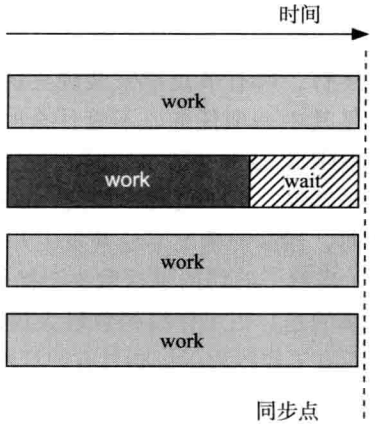


图 5-14 少数线程为“speeder”的负载不均衡现象（这里只有一个），资源的低效利用有可能是可以接受的

导致负载不均衡的原因是多方面的，一般可分为两类：一是算法原因，应通过改进或者选择完全不同的算法来解决负载不均衡问题；二是优化原因，应通过修改完善代码解决负载不均衡问题。然而，有时候这两者又不容易区分：

- 所选择的在多个工作线程间分配任务的方法可能与问题结构不兼容。以 3.6.2 节讨论的分块 JDS 稀疏矩阵向量乘算法为例，在多线程任务分配时，可采用连续任务分配算法：即遍历整个循环（循环变量 ib），为每个线程分配连续的一组任务分块。由于 JDS 的存储机制，这样的任务分配（根据实际的矩阵结构）可能会导致负载不均衡。这是因为随循环计数器的增大，循环迭代在矩阵中的作用域越向下，相对应的对角线数量就越少。在这种情况下，使用循环任务分配甚至动态分配是比较好的选择。如编写共享内存并行程序，这将非常容易实现（参见 6.1.6 节）。
- 无论选用哪种并行方法（参见 5.2 节），在程序编译时可能无法确定处理一个工作块所需的时间。例如，为达到某收敛条件，需要每个工作线程执行一定数量的循环迭代。因为每个工作线程所要执行的迭代次数可能不相同，所以该算法本质就是负载不均衡的。 137
- 粗粒度问题可能会限制可用的并行性。当工作线程的数目明显不少于可分配的任务的数量时，这种情况经常会发生。开发额外的并行性（如果存在的话）可以缓解这类问题。

- 尽管任务的不均匀分配是导致负载不均衡的最主要原因，但是也存在着其他因素，如果一个工作线程不得不等待某一资源（如 I/O 或者通信设备），虽然这样的等待时间不能算作有效工作时间，但依然可以引起工作延迟，将这个工作线程转变为“lagger”（这里不能与操作系统抖动混淆，详细请参考下一节讨论）。除此之外，这种类型的开销往往具有统计性，引发不稳定的负载不均衡现象。

如果负载不均衡确定是影响性能的主要因素，则应该考虑是否有不同的任务分配策略可以消除（至少减少）负载不均衡现象。当完全均衡的任务分配不能实现时，消除“lagger”可能足以大幅提高可扩展性。此外，通过重叠有效任务的执行来隐藏 I/O 和通信开销也是避
138 免负载不均衡的方法 [A82]。

操作系统抖动

近期在大规模商业并行系统构建时，发现了一个特殊而有趣的导致负载不均衡的原因，这个原因会引发令人意外的结果 [L77]。最标准的分布式存储并行计算机的安装方法为各节点独立运行，所有节点都安装独立的操作系统。操作系统会负责很多日常工作，运行用户程序只是其中一项任务。不管什么时间，操作系统的常规任务如写日志文件、提供性能指标、清空磁盘 cache 区、启动作业系统等，会抢占运行资源，一个正在运行的应用程序进程可能会被延迟。由于负载不均衡，这个“lagger”将在下一同步点使并行程序的执行略有延迟。然而，当这种情况不经常发生并且进程数量非常少时，这个延迟是可以忽略的（参见图 5-15a）。当然，确切的延迟取决于操作系统的活动周期和同步频率。

不幸的是，当工作线程数量大规模增加时，情况就会发生变化。这是因为“操作系统噪声”对所有工作线程的影响具有统计性。工作线程数量越多，两个连续同步点间发生延迟的可能性就越高。当代码中同步点的频率和噪声引发延迟的平均频率接近时，负载不均衡现象将会频繁发生。这个现象也称“共振”[L77]。图 5-15b 展示了这种现象的一个缩略场景。注意，在大规模并行系统中，这个性能限制是非常严重的，因为实际应用程序面对的不仅是几十或者几百个计算节点。在一些小规模系统中也存在性能抖动现象，但它们与操作系统抖动无关。

除尽可能减少操作系统活动（如关闭未使用的守护进程、轮询和日志功能，或者每个节点设一个专用处理器处理操作系统任务）外，减少操作系统抖动的一个有效方法是为所有的工作线程同步不可避免的、周期性的操作系统活动（参见图 5-15c）。即在一个相同同步点对齐所有工作线程延迟，这种情况下，性能损失不会大于情况 a。然而，这些不是标准的处理方法，而且需要对操作系统做本质的改变。尽管如此，随着大规模并行计算机节点和核数的增多，消除操作系统噪声可能很快成为这些大规模并行计算机的共同特点。

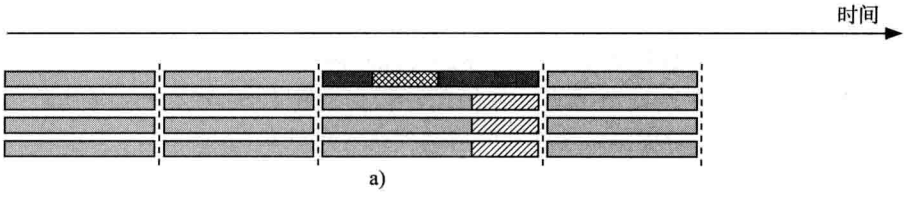
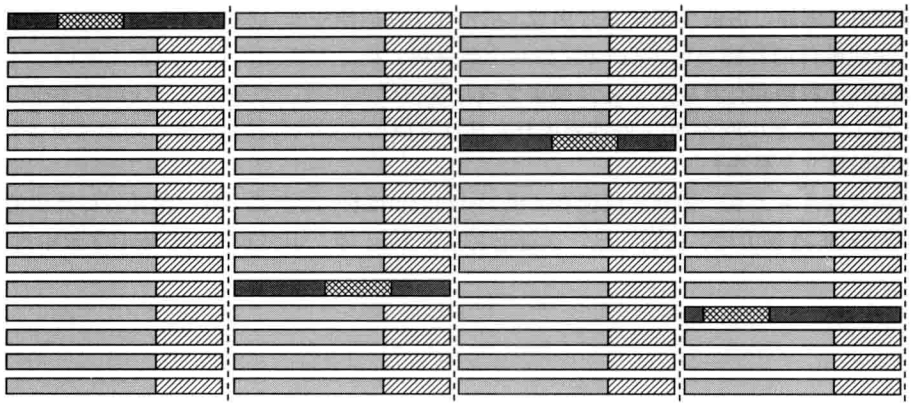
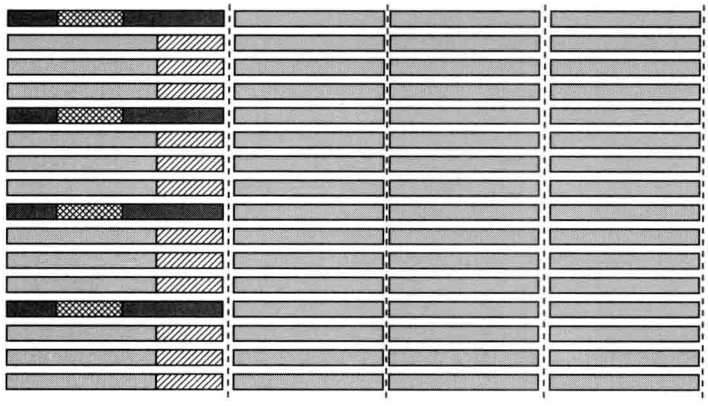


图 5-15 假设操作系统相关延迟（交叉阴影区域）以一定概率发生。当工作线程数目非常少时，对并行性能的影响可能也非常小 a）；增加线程数目（弱可扩展性场景下），会增加在下一个同步点前操作系统延迟发生的可能性，从而延长整体运行时间 b）；机器上的所有操作系统都同步操作系统行为以消除“操作系统抖动”，可提高程序性能 c）（图片改编自 [L77]）



b)



c)

图 5-15 （续）

习题

5.1 通信与计算重叠。如果通信和计算可以重叠（假设硬件支持并且制定了相应的算法），如何对5.3.8节讨论的性能较差的处理器的强可扩展性分析做本质改变？要考虑如果通信时间超过计算时间，两者不会完美重叠的情况。

139
~
140

5.2 选择最优的工作线程数量。如果一个应用程序的可扩展性表现为性能饱和或者随 N 的增加反而下降，就会随之产生选择最优工作线程数量的问题。在这种情况下，程序员不会去选择性能最优点（或者接近饱和点），因为并行效率已经非常低了。真正需要的是一个不鼓励使用过多工作线程的“成本模型”。绝大多数计算中心以“CPU 的时钟时间”作为计算时间收费，即使用 N 个 CPU 运行时间为 T_w，总计费时间与 NT_w 成比例。对于用户来说，最小化时钟时间（即解决问题时间）和成本需要提供一个合理的平衡。假定通信开销一定（即延迟受限情况）的强可扩展性分析，请给出最优工作线程数量 N_{opt} 的一个条件。N_{opt} 个工作线程会带来什么加速化？

5.3 同步操作的影响。同步所有的工作线程是非常耗时的，因为同步操作引发的成本与工作线程数量的比例在对数和线性之间。对于强扩展性和弱扩展性，同步的影响分别有哪些？

5.4 加速设备。当前，为标准计算节点配备加速设备越来越流行。这个想法的一个常见改变是在标准

计算节点（例如，由两个多核芯片组成）的一个 I/O 插槽中安装特定硬件。对于执行某些特定操作，加速硬件的性能要比主机 CPU 高几个数量级，但是其可用内存一般要比主机 CPU 小得多。将应用程序移植到计算设备上需要确定要进行移植代码片段。如果这部分代码在加速设备上的加速比是 α，要至少取得 90% 的性能提升，请计算原始代码的移植量（移植到加速设备）？限制内存大小的意义是什么？

141 ~ 142

5.5 用性能数据愚弄大众。强烈建议读者阅读 David H. Bailey 的幽默文章“用并行计算机的性能结果愚弄大众的 12 种方法”[ST]。尽管这篇论文写于 1991 年，但是很多观点依旧非常中肯。

使用 OpenMP 进行共享存储并行编程

在多核时代，单槽单核系统除在嵌入式市场还存在，但在其他领域已经基本消失。性价比这一关键因素主要用来衡量每个槽有多个核（可能为多个芯片）的双槽机制中。尽管没有共享内存的概念依然可以运行多进程程序，但是基本的并行编程理应从共享内存角度出发。请参考第 9 章的分布式存储并行编程的相关内容。

共享内存编程并非多核时代的发明。多处理器（单核）系统已经出现数了十年，而合适的可移植编程接口也在 20 世纪 90 年代被开发出来，其中最出名的是 POSIX 线程。基本的共享内存并行编程的限制和瓶颈及其他的并行模型类似（请见第 5 章）而其与众不同的特性将在第 7 章阐述。本章的目的是大致介绍如今最主流的共享内存编程标准 OpenMP。OpenMP 自当前标准版（3.0）起已用 C、C++ 和 Fortran 语言实现。一些用来优化的 OpenMP 结构将主要在第 7 章介绍。

需要补充的是，有些用 C++ 实现的特有解决方案在某些方面提供了比 OpenMP 更好的功能，例如英特尔线程构件（Intel Threading Building，TBB）[P10]。鉴于目前基于编译器的自动共享内存并行化除了一些常见情况之外无法达到预期效果，我们将特意忽略它。

6.1 OpenMP 简介

共享内存能够提供所有处理器及时访问所有数据的可能性而避免显式通信。不幸的是，对于科学软件尤其是循环中心程序，POSIX 线程并不是一个合适的并行编程模型。由于这个原因，编译器供应商共同努力确立共同标准，即是 OpenMP。OpenMP 就是一系列的编译器指令，对于不支持 OpenMP 的编译器来说这些指令将被当作注释忽略。因此，一个良好的并行 OpenMP 程序应该也是一个合法的串行程序（这当然不是强制要求，但是它极大的有助于简化开发和调试）。一个 OpenMP 程序核心实体是线程而非进程。因为几个线程能够共享一块公共地址空间和互斥访问数据，所以线程又称为“轻量级进程”。相对于创建一个进程，开启一个线程的代价更小，这是因为除了程序计数器、指令指针（下一个即将执行的指令的地址）栈指针和寄存器状态外，线程将共享一切。每一个线程通过局部栈指针能够拥有“私有”的变量，但是既然所有的数据可以通过共享内存空间访问，那么所有其他线程只需要获取该变量指针便可访问。但是，OpenMP 标准实际上禁止其他线程访问线程的私有对象。随后我们可以清晰地看出这是个好的想法。

我们将集中精力于 OpenMP 的 Fortran 接口，并且会在合适的时候指出它与 C/C++ 接口重要的区别。

6.1.1 并行执行

任一 OpenMP 程序启动后，一个单线程即主线程会立即运行。并行执行真正存在于任意数量的并行区域中。在两个并行区域之间，除了主线程执行代码外，其他任何线程将不执

行。这就是所谓的“fork-join 模型”（请参考图 6-1）。在一个并行区域里，一个线程组并发地执行指令。不同的并行区域可能有不同数量的线程。

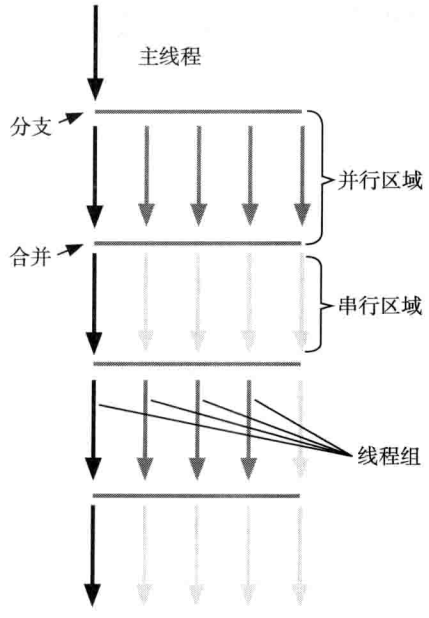


图 6-1 OpenMP 线程模型操作：主线程“forks”一组作用于共享内存的并行区域的线程。并行区域结束后，这组线程“joined”一起，也就是，终止或者挂起直到先一个并行区域开始。不同并行区域的线程数可能不同

144 OpenMP 是将原始操作系统线程接口进行适配以使其拥有更易于被数值软件典型结构采用的层次。在实践中，Fortran 并行区域分别被 !$OMP PARALLEL 所初始化和被 !$OMP END PARALLEL 所结束。!$OMP 字符串即是所谓 OpenMP 指令开始的标记（在 C/C++ 中使用 #pragma omp）。在一个并行区域中，每一个线程包含一个唯一标识符，即它的线程 ID，取值为零到线程总数减一，并且能够被 omp_get_thread_num() API 获取。

```
1   use omp_lib       ! module with API declarations
2
3   print *,'I am the master, and I am alone'
4  !$OMP PARALLEL
5   call do_work_package(omp_get_thread_num(),omp_get_num_threads())
6  !$OMP END PARALLEL
```

omp_get_num_threads() 函数返回当前并行区域中的活跃线程总数。omp_lib 模块包含 API 定义（分别有 Fortran 77 和 C/C++ 实现的包含文件 mpif.h 和 omp.h）。在 OMP PARALLEL 和 OMP END PARALLEL 之间的代码，包括子过程调用，这个子过程被每一个线程调用。最简单的情况是，线程 ID 用来区别不同线程所执行的任务；上面例子中是通过将子过程线程 ID 和线程总数作为参数调用 do_work_package() 实现。采取这种方式的 OpenMP 等价于 POSIX 线程编程模型。

Fortran 和 C/C++ 的 OpenMP 之间一个重要的区别需要在这里强调。在 C/C++ 中，没有 end parallel 指令，这是因为所有的指令应用于随后的语句或者结构块。上面例子在 C++ 中实现如下：

```
#include <omp.h>

std::cout << "I am the master, and I am alone";
#pragma omp parallel
{
  do_work_package(omp_get_thread_num(),omp_get_num_threads());
}
```

花括号在此情况下可以省略，但是事实上一个结构块隶属于对数据域有影响的并行执行（参考下面）。

实际运行的线程数量在编译时并不确定。在运行之前可以通过环境变量设置：

```
$ export OMP_NUM_THREADS=4
$ ./a.out
```

尽管通过程序控制可以设置和改变运行的线程数量，但是编写一个 OpenMP 程序不应该对线程的具体数量进行假定。 145

代码清单 6-1 “Manual”循环并行化和变量私有化。注意这并不是 OpenMP 程序特有的模型

```
  integer :: bstart, bend, blen, numth, tid, i
  integer :: N
  double precision, dimension(N) :: a,b,c
  ...
!$OMP PARALLEL PRIVATE(bstart,bend,blen,numth,tid,i)
  numth = omp_get_num_threads()
  tid = omp_get_thread_num()
  blen = N/numth
  if(tid.lt.mod(N,numth)) then
    blen = blen + 1
    bstart = blen * tid + 1
  else
    bstart = blen * tid + mod(N,numth) + 1
  endif
  bend = bstart + blen - 1
  do i = bstart,bend
    a(i) = b(i) + c(i)
  enddo
!$OMP END PARALLEL
```

6.1.2 数据作用域

存在并行区域之前的任何变量都将在并行区域内部被访问，同时默认被所有线程共享。当且仅当每个线程拥有自己的私有变量才使得共享工作有意义。OpenMP 通过为每一个线程定义单独的栈来实现这一概念。有 3 种方式实现私有变量：

1）处于并行结构入口之前的变量可被私有化，即可以通过在 OMP PARALLEL 指令添加 PRIVATE 语句为每一个线程构造属于它自己的变量作为私有实例。这个私有变量的作用域持续到并行结构结束。

2）共享循环（参考下一节）的指示变量自动变为私有变量。

3）被并行区域调用的子过程局部变量对每个调用线程来说是私有变量。这也适用于使用值传递语义产生的实参拷贝情况和 C/C++ 结构块中声明的变量。但是，在 Fortran 中具有 SAVE 属性（或者 C/C++ 中具有 static 属性）的局部变量将被共享。

在并行区域不被修改的共享变量不用私有化。

代码清单 6-1 展现了两个数组相加的简单循环的并行化。实际的循环位于 16 ～ 18 行， 146

在此之前的代码只是计算每个线程的循环边界。第5行PARALLEL指令中的PRIVATE语句将所有确定的变量为私有化，即每个线程从它们的局部栈获取每个变量没有初始值的实例（C++对象被默认构造函数初始化）。使用FIRSTPRIVATE指令替代PRIVATE将会使用共享实例内容（在C++中，将使用拷贝构造函数）来初始化私有化实例。在并行区域之后，如果私有化变量不被刻意改变，那么它们的最初值将会保留。注意这里有单独语句（分别为THREADPRIVATE和COPYIN[P11]）用于全局或者静态数据的私有化（SAVE变量、常用块元素、静态变量）。

在C/C++中，由于parallel指令应用于结构块，所以多数情况下没有必要使用private语句。相对于私有化共享实例，我们也可以只声明局部变量。

```
1 #pragma omp parallel
2 {
3   int bstart, bend, blen, numth, tid, i;
4   ...                 // calculate loop boundaries
5   for(i=bstart; i<=bend; ++i)
6     a[i] = b[i] + c[i];
7 }
```

如上所示的人工循环并行化当然不是OpenMP特有的操作模型。OpenMP标准定义了关于线程间分布式任务更为高级的方法。

6.1.3 循环的OpenMP工作共享

循环作为科学计算代码中无所不在的编程结构，如果它的每次迭代独立，那么通常会作为并行化的候选。这对应着5.2.1节中描述的中等粒度的数据并行。作为例子，考虑一个积分计算 π 的简单并行程序：

$$\pi = \int_0^1 dx \frac{4}{1+x^2} \tag{6-1}$$

代码清单6-2展示了一个可能的实现。对比前面的例子，这个程序也是一个合法的串行代码。通过PARALLEL指令中的FIRSTPRIVATE从句，sum初始值被复制到私有实例。然后，在do循环之前的DO指令开启一个worksharing结构：循环的迭代被分布在各个线程上（这是因为处在并行区域中）。每个线程获取它们自己的迭代空间，即i的值被赋值为不同的数集。如何将线程映射到迭代默认是依赖于实现的，但是程序员可以控制（参考下面的6.1.6节）。尽管循环计数器i在并行区中被共享，但是它自动私有化。循环后的最后的END DO
147 指令在这里并不是严格需要的，但是在NOWAIT语句出现的时候必须使用；详细情况请参考7.2.1节。一个DO指令后面必须跟随一个do循环，并且只作用于这个循环。在C/C++中，具有同样功能的是for指令。循环计数器只能是整数（有符号或者无符号）、指针，或者随机访问迭代器。

代码清单6-2 OpenMP中函数数值积分的一个简单程序

```
1   double precision :: pi,w,sum,x
2   integer :: i,N=1000000
3
4   pi = 0.d0
5   w = 1.d0/N
6   sum = 0.d0
7 !$OMP PARALLEL PRIVATE(x) FIRSTPRIVATE(sum)
8 !$OMP DO
```

```
9    do i=1,n
10     x = w*(i-0.5d0)
11     sum = sum + 4.d0/(1.d0+x*x)
12   enddo
13 !$OMP END DO
14 !$OMP CRITICAL
15   pi= pi + w*sum
16 !$OMP END CRITICAL
17 !$OMP END PARALLEL
```

在并行循环中，每一个线程执行整个循环迭代空间中属于它自己的部分，将结果累加到私有变量 sum（11 行）。循环结束之后，仍旧处在并行区域中，而部分和必须加到最终的结果（15 行），这是因为 sum 私有实例在并行区域结束后消失。这里存在一个问题：在没有任何计数度量的情况下，线程将并发的将结果写回 pi。因此，计算结果将依赖于线程访问 pi 的顺序，而且大多情况顺序是错误的。这就是竞争条件（race condition），我们将在下一个小节介绍如何阻止它出现。

循环的工作共享，即使并行循环被并行区域的子程序调用，也会工作。因为 DO 指令超出了并行区域词法范围，所以这时 DO 指令被孤立。如果一个指令没有遇到一个活动并行区域，则循环将不会被工作共享。

最后，如果一个共享工作循环的并行区域分离不是必须的，那么可以将两条指令合并。

```
1 !$OMP PARALLEL DO
2   do i=1,N
3     a(i) = b(i) + c(i) * d(i)
4   enddo
5 !$OMP END PARALLEL DO
```

合并循环共享工作指令的语句集合其实是所有单独指令语句的合集。 148

6.1.4 同步

1. 临界区

并发地访问共享变量或者一般的共享资源必须尽量避免竞争条件。临界区（critical region）通过保证某一时刻最多只有一个线程执行一段代码来解决这个问题。如果一个线程在执行临界区域代码，这时另一个线程想要进入，则第二个线程必须等待（阻塞）直到先前的线程离开临界区。在上面积分的例子中（代码清单 6-2），CRITICAL 和 END CRITICAL 指令（14 行和 16 行）包括其 pi 来保证结果的正确性。需要注意，进入临界区的线程顺序是不确定的，并且每次程序运行都不同。因此，“正确结果”的定义必须包含按照随机顺序累加的部分和的可能性，当然关于浮点数精度的通常保留建议也适用 [135]。（如果强顺序的等值，即和串行代码类似的按位相等是必须的，那么 OpenMP 提供了 ORDERED 结构体的一种可能解决方案，这里我们不再赘述。）

如果使用临界区不当将会造成死锁的危险。当一个或者多个“代理”（这里指的是线程）等待着永远不能获取的资源，那么死锁将会产生，其中一个情形是由不合理的排列 CRITICAL 指令造成的。当一个线程在临界区碰到 CRITICAL 指令将会永远阻塞。由于这可能出现在多层嵌套的子过程中，所以死锁一般很难确定。

OpenMP 提供了一种解决这种问题的方案：我们可以为一个临界区取一个区别于其他临界区的名字。名字通过在 CRITICAL 指令后的括号中指定。

```
1  !$OMP PARALLEL DO PRIVATE(x)
2    do i=1,N
3      x = SIN(2*PI*x/N)
4  !$OMP CRITICAL (psum)
5      sum = sum + func(x)
6  !$OMP END CRITICAL (psum)
7    enddo
8  !$OMP END PARALLEL DO
9    ...
10   double precision func(v)
11   double precision :: v
12 !$OMP CRITICAL (prand)
13   func = v + random_func()
14 !$OMP END CRITICAL (prand)
15   END SUBROUTINE func
```

在第5行的sum更新是受临界区保护的。因为不允许多于一个线程同时调用random_func()（13行），所以在子过程func()中存在另一个临界区；很有可能随机种子是SAVE属
149 性变量。这样的函数是非线程安全的，即它并发的执行可能造成竞争条件。

如果不同的临界区没有名字，那么代码将会死锁，这是因为调用func()的线程已经处在临界区，这将导致立即碰到第二个临界区并且自己永远等待自己释放资源。如果有不同的名字，那么第二临界区将被用来保护不同于第一个临界区保护的资源。

命名临界区的缺点就是名字必须是唯一标识符。例如，不能将它们用整数变量来索引。在OpenMP API函数中提供锁（lock）来保护共享资源的方式。使用锁变量的好处是它可以放在数组或者结构体中。因此，这种方式可以单独保护数组中的每一个元素而不需要在编译时知道它们的编号。请参考7.2.3节的实例。

2. 同步点

当并行执行的某一个点需要同步所有的线程时，可以使用BARRIER指令。

```
1  !$OMP BARRIER
```

这个同步点是一个保证所有线程在其他线程执行它下面的代码之前都必须到达的点。当然必须保证其他线程都必须到达同步点，否则会发生死锁。

在OpenMP程序中同步点的使用需要谨慎，部分是因为会引起死锁的可能性，还有就是影响性能（同步需要开销）。注意每一个并行区域的结尾都有一个不可删除的隐形同步点。在工作共享循环结束和一些阻止竞争条件的结构体中都会有默认的隐形同步点。它们可以用NOWAIT语句删除。详情请参考7.2.1节。

6.1.5 归约

代码清单6-3中展现的循环代码是用来为数组a()加上一些随机噪声，然后计算它的向量范数。RANDOM_NUMBER()子过程根据OpenMP保准被假定是线程安全的。

类似于代码清单6-2中的积分代码，循环实现了归约操作：许多的贡献（a()中元素的更新）被累加到一个变量。前面我们用临界区解决这个问题，但是OpenMP提供了使用REDUCTION语句支持归约的更简便选择（第5行末）。它自动地私有化确定的变量（这里是s），并且将私有的变量实例初始化为合理的值。在这个结构的末尾，所有的部分结果都使用特殊操作符（这里是+）被累计到一个共享实例s中从而获取最终结果。

OpenMP归约支持一些操作符集合（Fortran和C/C++略有不同），这些集合不可扩展。
150 C++重载操作符也是不允许的。但是最常见的情形（加、减、乘、逻辑等）都包含其中。如

果一个操作符没有定义，那么必须使用代码清单 6-2 中的“手动”方法。

代码清单 6-3 为数组元素添加噪声并计算向量范数的归约语句例子

```
1   double precision :: r,s
2   double precision, dimension(N) :: a
3
4   call RANDOM_SEED()
5  !$OMP PARALLEL DO PRIVATE(r) REDUCTION(+:s)
6   do i=1,N
7     call RANDOM_NUMBER(r)   ! thread safe
8     a(i) = a(i) + func(r)   ! func() is thread safe
9     s = s + a(i) * a(i)
10  enddo
11 !$OMP END PARALLEL DO
12
13  print *,'Sum = ',s
```

注意归约变量的自动初始化尽管方便易用，但是可能产生不合法的序列，也就是非 OpenMP 代码。不使用 OpenMP 编译上面的例子，s 将不会被初始化。

6.1.6 循环调度

正如上面所述，循环迭代到线程的映射是可配置的。它被共享工作循环指令中的 SCHEDULE 语句参数所控制。

```
1 !$OMP DO SCHEDULE(STATIC)
2   do i=1,N
3     a(i) = calculate(i)
4   enddo
5 !$OMP END DO
```

最简单的可能是 STATIC，它将循环分成连续、等大小（近似的）的组块。每一个线程仅执行其中的一块。如果一些循环迭代的工作量不是固定的，也就是随着循环变量递减，那么这种方法是次优的，这是因为不同的线程将会获得差别很大的负载，这将会导致负载不均衡。一个解决方案就是在“ STATIC,1”中使用 chunksize 指定每个组块的大小为 1，并使得每个线程循环获取。块大小可以不是常数而是任何合法整数值的表达式。

除了静态调度，还存在其他类型的负载（参见图 6-2）。动态调度是为下一个已经完成自己块组的线程分配一个通过 chunksize 定义的块组工作。这样会产生每一次运行都不一样的 151
弹性分配。被分配“更简单”块组的线程将会在结束时完成更多，这样负载不均很大程度会被削弱。

如果相对于执行时间来说块组太小，那么动态调度将会产生巨大开销（参考 7.2.1 节的循环开销估计）。这也是为什么对于那些导致负载不均的循环选取较大的块组。当遇到问题的时候，使用诱导式调度会有帮助。同样，线程动态地获取块组，但是块组的大小总是正比于剩余迭代次数除以线程数的值。最小的块组大小是由调度语句确定的（默认是 1）。除了动态分配块组，调度开销也是受控的。但是，考虑动态调度和诱导式调度需要注意：由于分配给线程块组是动态不确定的，所以受带宽限制的应用程序在 ccNUMA 系统中将会受限于访存局部性（参考 4.2.3 节的 ccNUMA 体系结构和第 8 章的 ccNUMA 性能影响和优化方法）。如果使用标准的工作共享指令，那么在这种情形下只能使用静态调度。当然，还存在 152
“显式”调度可能性，6.1.2 节中使用线程 ID 来给线程分配工作。

关于调试和程序概要分析，OpenMP 提供了运行时决定循环调度的方法。如果在调度语

句中指明“RUNTIME”，那么循环将会根据 OMP_SCHEDULEshell 变量内容进行调度。但是，SCHEDULE(RUNTIME) 没有办法为不同的循环设置不同的调度方式。

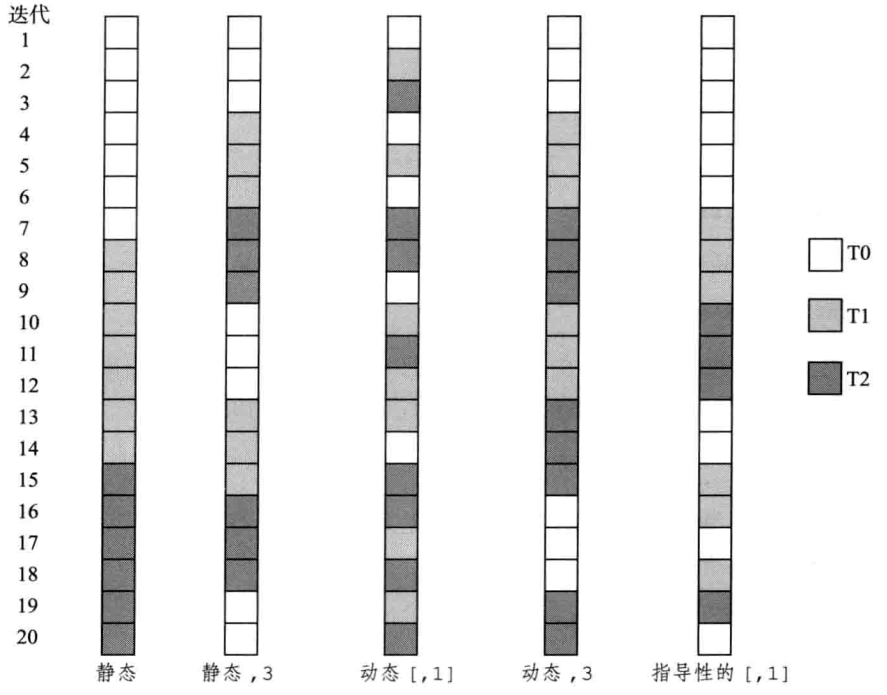


图 6-2 OpenMP 中的循环调度。这个例子中循环有 20 次迭代，由三个线程（T0、T1、T2）执行。DYNAMIC 和 GUIDED 默认块组是 1。如果一个块组大小确定，那么最后一个块组可能要短一些。注意，STATIC 调度保证块组的分配在每次运行的时候都相同

6.1.7 任务

在早期的标准中，OpenMP 中的并行工作共享主要针对于循环结构。但是，并不是所有并行工作都是循环形式；一个典型的例子就是应该并行处理存放着对象的线性链表（很可能存放在 STL 容器 std::list<> 中）。既然通过整数索引或者随机访问迭代器不是很容易定位链表元素，那么循环工作共享结构就不适用了，否则将需要大量的编程努力才能使用。

OpenMP 3.0 提供了任务的概念来规避这种限制。由 TASK 指令定义任务，并且包含要被执行的代码。[⊖] 当线程碰到一个任务结构体，线程可能立即执行该结构体或者设置合适的数据环境并延迟执行。当任务准备好了，它将被小组的其他线程执行。

作为一个简单例子，考虑这样一个循环，其中的函数调用是根据每次循环索引计算的概率决定：

```
1   integer i,N=1000000
2   type(object), dimension(N) :: p
3   double precision :: r
4   ...
5  !$OMP PARALLEL PRIVATE(r,i)
6  !$OMP SINGLE
7   do i=1,N
```

⊖ 在 OpenMP 术语中，“任务”实际上是一个通用术语；这里的定义足以满足我们需要。

```
8       call RANDOM_NUMBER(r)
9       if(p(i)%weight > r) then
10  !$OMP TASK
11        ! i is automatically firstprivate
12        ! p() is shared
13        call do_work_with(p(i))
14  !$OMP END TASK
15      endif
16    enddo
17  !$OMP END SINGLE
18  !$OMP END PARALLEL
```

执行 do_work_with() 的实际次数是不确定的，所以任务化是一个自然的选择。遍历所有 p() 元素的 do 循环是在一个 SINGLE 区域执行的（6 ～ 17 行）。一个 SINGLE 区域将只能被一个线程进入，也即允许最先到达 SINGLE 指令的线程进入。所有其他线程将会跳过这 153
段代码直到 END SINGLE 指令，并在那里的一个隐式同步点等待。线程将会根据当前具体对象内容以一定概率进入一个任务结构。一个任务存包括 do_work_with()（第 13 行）调用以及合适的数据环境，该环境是由 p() 类型的数组和索引 i 组成。当然，每个任务的索引是唯一的，所以它实际上是由 FIRSTPRIVATE 语句决定。OpenMP 明确指出处在封闭上下文中的私有变量在任务中将被自动变为 FIRSTPRIVATE，然而共享的数据依旧共享（除了添加额外的数据作用域）。由于这正是我们需要的情况，所有不需要额外增加语句。

在 SINGLE 区域中的线程生成的所有任务都将受整个线程组的动态执行控制。实际上，生成线程也可能被强制挂起 TASK 结构体的循环执行（任务调度点的一个实例）以便运行排队的任务。当排队的任务到达内部限制（依赖实现）时，这种情形将发生。在一些任务已经运行后，生成线程将会返回循环。注意，任务调度的复杂性这个简单例子不能解释透彻；多线程能够并发地生成任务，并且任务可以被声明为 untied，以便不同线程在任务调度点开始执行。OpenMP 标准提供了大量的实例。

当动态或者诱导式循环调度时，任务并行的不确定执行将导致和 ccNUMA 局部访问的同样问题。改善这种问题的编程技巧确实存在 [O58]，但是它们的应用有限。

6.1.8 其他方面

1. 条件编译

在一些情形中，编写依赖 OpenMP 打开或关闭的代码将很有用。指令本身不存在什么问题，这是由于它们将被自动忽略。对于默认情况，有些人想屏蔽掉，例如，如果没有开启 OpenMP，那么 API 函数的调用或者其他相关代码都没有意义。C/C++ 将通过预处理符号 _OPENMP 支持，但也仅仅是在 OpenMP 存在的情况。在 Fortran 中如果 OpenMP 关闭，那么标记“!$”将作为注释。

2. 存储一致性

在代码清单 6-4 展示的代码中，第二个 API 调用（第 8 行）处在 SINGLE 区域。这是因为 numthreads 是全局的并且应该只被一个线程写入。在临界区，每一个线程将会打印一条消息，但是必须保证 numthreads 变量的改变值被“提交”到内存之前没有一个线程离开 SINGLE 区域。END SINGLE 指令这里作为隐式的同步点，即其他线程达到该点前，没有线程能够继续执行。OpenMP 内存模型确保同步点强制内存一致性：保留在寄存器中的变量写 154
入内存以便 cache 一致性能够保证所有 cache 将获得最新更新值。这可以通过在程序控制中使用 FLUSH 指令控制，但是大多数 OpenMP 工作共享和同步结构体执行隐式同步，依旧是

在结尾刷新。

代码清单 6-4 OpenMP 中 Fortran 标记和条件编译混合

```
1  !$ use omp_lib
2    myid=0
3    numthreads=1
4  #ifdef _OPENMP
5  !$OMP PARALLEL PRIVATE(myid)
6    myid = omp_get_thread_num()
7  !$OMP SINGLE
8    numthreads = omp_get_num_threads()
9  !$OMP END SINGLE
10 !$OMP CRITICAL
11   write(*,*) 'Parallel program - this is thread ',myid,&
12                                        ' of ',numthreads
13 !$OMP END CRITICAL
14 !$OMP END PARALLEL
15 #else
16   write(*,*) 'Serial program'
17 #endif
```

注意编译器优化可能阻止其他线程立即看见修改的变量。如果有疑惑，使用 FLUSH 指令或者声明变量为 volatile（只在 C/C++ 和 Fortran 2003 中可行）。

3. 线程安全

因为第 11 行的 write 语句是串行的（即被临界区保护），所以当多线程写到控制台时，它的输出将不会打乱。作为一般性原则，不仅 I/O 操作及很一般的操作系统功能，还有通用的库函数都应该是串行的，否则线程不安全。一个突出实例就是 C 语言库函数 rand()，它用静态变量来存储隐藏状态（种子）。

4. 亲缘性

需要注意的是 OpenMP 标准没有给出线程如何在系统中绑定内核，也没有给出局部性约束的实现。所以对于线程的安排，我们不能依靠操作系统来做出正确的决策，这也是为什么（尤其在多核架构和 ccNUMA 系统中）要使用系统级的工具、支持编译器或者库函数显
155 式将线程绑定到内核。更多技术细节见附录 A。

5. 环境变量

OpenMP 程序某些方面的执行可能受到环境变量的影响。正如上面描述的 OMP_NUM_THREADS 和 OMP_SCHEDULE。

对于线程的局部变量，读者必须时时牢记一般情况下操作系统 shell 限制进程总的栈变量的大小，并且系统也对每个线程的栈大小有限制。这种限制可以通过 OMP_STACKSIZE 环境变量进行调整。例如，设置它为“100M”将会将每个线程的栈大小置为 100MB（除去由 shell 设置的初始程序线程栈）。栈溢出是 OpenMP 程序常见的问题。

OpenMP 标准允许不同的并行区域之间可以动态改变活跃线程数量以便适应可获取系统资源（动态线程数量调整）。这个特性可以设置 OMP_DYNAMIC 环境变量真或者假来打开或者关闭。它没有明确 OpenMP 运行时实现的默认行为。

6.2 案例分析：OpenMP 并行实现 Jacobi 算法

在 3.3 节 Jacobi 算法并行化采取的是直接的方式。但是我们将加上一个细微的改动：一个收敛标准将确保代码产生一个收敛的结果。为了做到这样，我们引进一个新的变量 maxdelta 来存放所有网格中前后迭代的最大绝对差值（参考代码清单 6-5）。如果 maxdelta

降到域值 eps 以下，那么达到收敛。

幸运的是，OpenMP Fortran 接口允许在 REDUCTION 语句中使用 MAX() 原语函数来简化收敛检查（代码清单 6-5 的第 7 和 15 行）。图 6-3 显示了在 Intel 双槽 Xeon 5160 3.0 GHz 节点上一个、两个和四个线程的性能。在这个节点上，一个槽中的双核共享 4MB L2cache 和一个前端总线（FSB）。结果证实了多核环境下的并行计算的几个关键方面：

- 随着 N 的增加，当工作集大小（$2\times N^2\times 8$ 字节）超出 cache 时，性能将会下降。对于同样的 N，性能下降也会发生在单线程和运行在同一个 L2 组（被填充符号）中的双线程。如果线程处在不同的槽（开放符号），这个限制将是$\sqrt{2}$倍大，这是因为合并的 cache 大小是双倍（图 6-3 中的虚线）。第二个下降点在 N 非常大的情况下，即当两个连续的网格行超出 L2cache 大小，就不能看作我们使用了方形网格（参考 3.3 节）。

代码清单 6-5　增加了收敛标准的 $N\times N$ 网格大小的二维 Jacobi 算法 OpenMP 实现

```
1    double precision, dimension(0:N+1,0:N+1,0:1) :: phi
2    double precision :: maxdelta,eps
3    integer :: t0,t1
4    eps = 1.d-14        ! convergence threshold
5    t0 = 0 ; t1 = 1
6    maxdelta = 2.d0*eps
7    do while(maxdelta.gt.eps)
8      maxdelta = 0.d0
9  !$OMP PARALLEL DO REDUCTION(max:maxdelta)
10     do k = 1,N
11       do i = 1,N
12          ! four flops, one store, four loads
13          phi(i,k,t1) = (  phi(i+1,k,t0) + phi(i-1,k,t0)
14                         + phi(i,k+1,t0) + phi(i,k-1,t0) ) * 0.25
15          maxdelta = max(maxdelta,abs(phi(i,k,t1)-phi(i,k,t0)))
16       enddo
17     enddo
18 !$OMP END PARALLEL DO
19     ! swap arrays
20     i = t0 ; t0=t1 ; t1=i
21   enddo
```

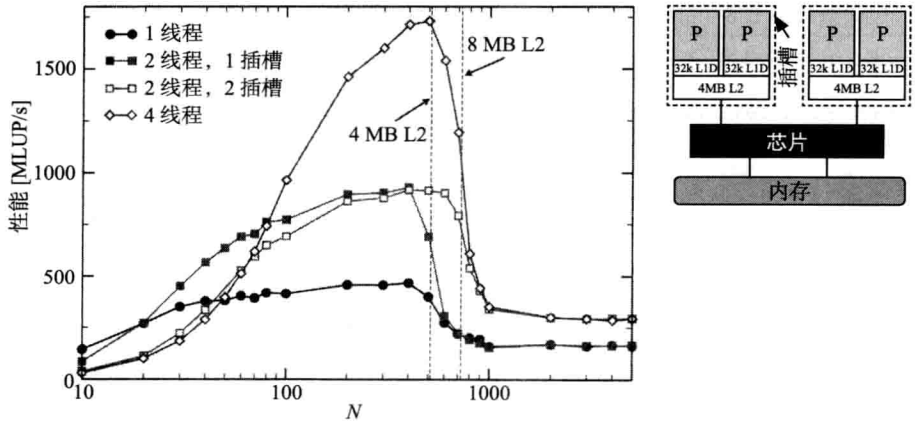


图 6-3　在 3.0 GHz Intel 双核双槽 Xeon 5160 节点（右边）上单个、两个和四个线程的 $N\times N$ 网格大小二维 Jacobi 算法 OpenMP 实现的不同问题规模与性能对比。对于两个线程，可以选择放在一个槽（实心方形）或者两个槽（空心方形）

156 ~ 157

- 对于带宽受限情况，单个线程可以充分利用一个槽的FSB，即比较大的N。同时在一个槽中运行两个线程没有性能上优势，这是因为它们竞争使用前端总线。增加第二个槽性能将会提升80%，很显然是因为两个FSB的缘故。因为芯片和FSB架构的缘故，扩展性并不是很好。注意在所有存储层次上的带宽扩展行为对于体系结构有很强的依赖性；这里存在使用两个或者多个线程用满内存接口的多核芯片。
- 当线程数目是两个时，最大的cache性能是一样的，不管这两个线程运行在相同或者不同的槽中（填充和开放方形对比）。这表明共享L2cache能够使组内双核的带宽需求饱和。然而，3/4的Jacobi内核加载由L1cache满足（参考3.3节的带宽需求分析）。这种情形下的性能分析很微妙[M41, M44]。
- 当$N < 50$时，线程位置对性能的影响要比数量重要，尽管这个问题适合于集成的L1cache。这种情形下用双槽速度下降一半。原因就是当N小时，OpenMP在工作共享循环的同步开销占主导地位。这个问题更详细的信息和如何减轻后果参考7.2节。

对于带宽受限算法的性能特点的解释需要对底层的硬件并行，包括多核芯片的问题都应该有一个好的理解。未来多核设计更加具有“差异性”（参考图1-17），并且展现多层cache组结构，使得对平行代码的性能特征理解造成困难。

6.3 高级OpenMP：波前并行化

直到目前，我们遇到的OpenMP并行问题或多或少都很直接，这是因为重要循环都是由独立的迭代组成。但是，在出现某些情况下阻止流水线的循环依赖中（参考1.2.3节），在循环之前编写一个简单工作共享指令将会导致不可预测的结果。一个典型的示例就是用来解决线性方程组或者边界值问题，同时作为多网格方法中平滑部分的Gauss-Seidel算法。代码清单6-6描述了三维空间中一个可能的串行实现。类似于3.3节中介绍的Jacobi算法，代码解决稳定状态，但是没有提供多余数组给当前和下一步使用；(i,j,k)的更改直接重用小坐
158 标中的三个相邻节点。因为那些已经在扫描之前更改，所以Gauss-Seidel算法拥有不同于Jacobi的收敛属性（Stein-Rosenberg理论）。

代码清单6-6 三维Gauss-Seidel算法的直接实现。高亮引用引起循环依赖

```
1 double precision, parameter :: osth=1/6.d0
2 do it=1,itmax ! number of iterations (sweeps)
3   !not parallelizable right away
4   do k=1,kmax
5      do j=1,jmax
6      do i=1,imax
7              phi(i,j,k) = ( phi(i-1,j,k) + phi(i+1,j,k)
8                           + phi(i,j-1,k) + phi(i,j+1,k)
9                           + phi(i,j,k-1) + phi(i,j,k+1) ) * osth
10         enddo
11       enddo
12    enddo
13 enddo
```

Jacobi算法的并行化很直接（参考前一节），这是因为一次扫描的更新存放在不同的数组，但是这里的情形不同。确实，在k循环之前加上PARALLEL DO指令会造成竞争条件并且每次运行产生的（错误）结果不同。

但是还是可以使用 OpenMP 并行化程序。关键想法就是找到一种能够穿越网格同时满足模板更新带来的依赖限制。图 6-4 和图 6-5 描述了如何实现这个方法：不再简单地将 k 维分割成块组给 OpenMP 线程处理，而是将一个 wavefront 传递给 k 个方向的网格。并行化的维度是 j，而 t 个线程 $T_0 \cdots T_{t-1}$ 中的每一个获取一个连续 j_{max}/t 大小的块组。这将网格分成 $i_{max} \times j_{max}/t \times 1$ 大小的块。k 坐标最小的第一个块仅能被单个线程（T_0）更新，形成自己的一个“wavefront”（波前，图 6-4 中的 W_1），其他所有线程不得不等待直到这个块完成。在那之后，第二个 wavefront（W_2）可以开始，这时两个线程（T_0 和 T_1）并行地处理两个块。在另一个同步之后，W_3 开启三个线程，以此类推。W_t 是首个用到所有线程的 wavefront，以所谓的 wind-up 步骤结束。在完全扫描之前需要一些时间（t 个 wavefront），wind-down 阶段开始并且工作线程数量在后续每个 wavefront 都会减少。k 和 j 坐标的最大值最后会被单线程（T_{t-1}）的第 W_n 个 wavefront 更新。在结束时，值为 $n = k_{max}+t-1$ 的 wavefront 已经以“流水线”的模式穿过网格。当然，$2(t-1)$ 只用少于 t 个线程。当 $k_{max} \gg t$ 时，整个设计才能到达负载均衡。

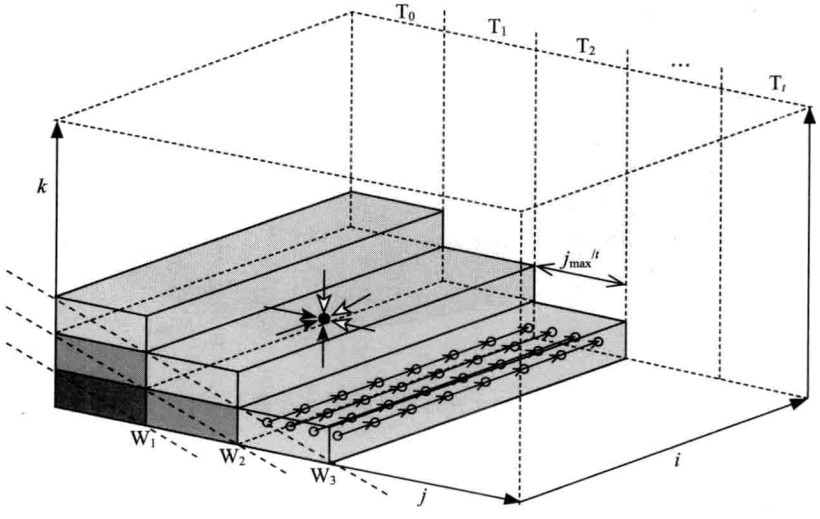


图 6-4 Gauss-Seidel 流水线处理（PPP），亦称三维 Gauss-Seidel 算法 wavefront 并行化（wind-up 步骤）。为了满足每次模板更新的依赖限制，连续的 wavefront（W_1，W_2，...，W_n）必须顺序执行，但是多线程每个 wavefront 中的多线程可以并行执行。到最后的 wind-up 步骤只有 t 个线程的子集参加

代码清单 6-7 显示了这个算法的一个可能实现。为了简化，我们假设 j_{max} 是线程数量整数倍。变量 1 计算 wavefront 个数，k 是每一个线程的当前 k 坐标。在第 19 行 OpenMP 的同步点使得所有线程（包括可能的空闲线程）在一个 wavefront 结束之前同步。 159

代码清单 6-7 三维 Gauss-Seidel 算法 wavefront 并行化。循环依赖依旧存在，但是线程能够并行工作

```
1  !$OMP PARALLEL PRIVATE(k,j,i,jStart,jEnd,threadID)
2    threadID=OMP_GET_THREAD_NUM()
3  !$OMP SINGLE
4    numThreads=OMP_GET_NUM_THREADS()
5  !$OMP END SINGLE
6    jStart=jmax/numThreads*threadID
7    jEnd=jStart+jmax/numThreads ! jmax is amultiple of numThreads
```

```
8    do l=1,kmax+numThreads-1
9      k=l-threadID
10     if((k.ge.1).and.(k.le.kmax)) then
11       do j=jStart,jEnd          ! this is the actual parallel loop
12         do i=1,iMax
13           phi(i,j,k) = ( phi(i-1,j,k) + phi(i+1,j,k)
14                        + phi(i,j-1,k) + phi(i,j+1,k)
15                        + phi(i,j,k-1) + phi(i,j,k+1) ) * osth
16         enddo
17       enddo
18     endif
19 !$OMP BARRIER
20   enddo
21 !$OMP END PARALLEL
```

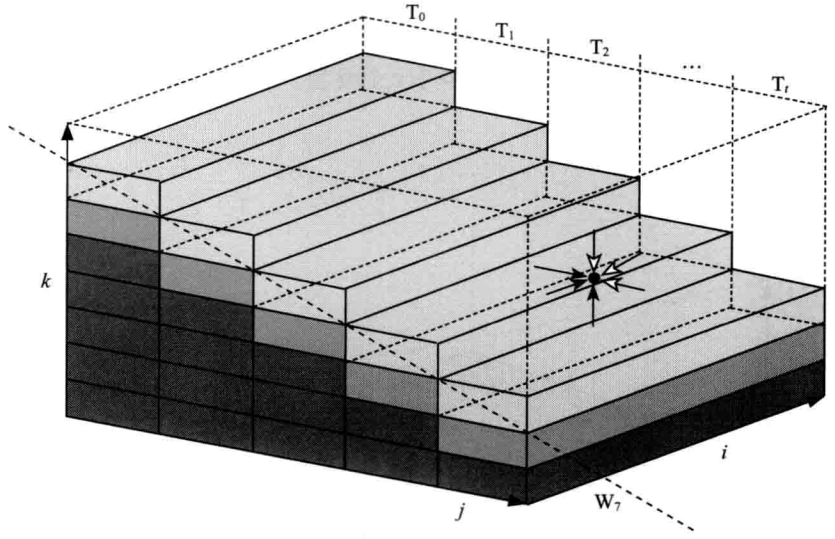


图 6-5 三维 Gauss-Seidel 算法 wavefront 并行化（满流水步骤）。所有 t 个线程都参加。第 W_7 个 wavefront 如实例所示

我们已经忽略诸如外层循环展开（参考图 6-4 的 T_2 块位置改变顺序）的标量优化。注意模板更新与先前的版本相比没有改变，所以这里仍然存在循环依赖。这些将阻止内存循环满流水的执行，但是如果性能受访存带宽限制，那么这样影响很小。实现流水线（也就是向量化）的可选方案参考习题 6.6。

wavefront 方法是高性能计算中最重要的方法之一，尤其适用大规模并行应用 [L76, L78] 和共享内存代码优化 [O52，O59]。wavefront 是流水线从中等粒度到粗粒度并行的一种自然扩展。遗憾的是目前主流的语言还没有对此的直接支持。进一步，尽管依赖分析是任何编译器的主要优化阶段，但很少有编译器能够做到自动 wavefront 并行化。

注意模板算法（Gauss－Seidel 和 Jacobi 算法仅仅是两个简单例子）是许多仿真和 PDE 解决方法的核心部分。过去几十年发明了许多优化、并行化以及向量化技术，并且存在大量的文献。更多的信息可以从参考文献 [O60, O61, O62, O63] 获取。

160
~
161

习题

6.1 OpenMP 代码纠错。下面的用 Fortran 90 编写的 OpenMP 并行代码哪里有错？

```
  double precision, dimension(0:360) :: a

!$OMP PARALLEL DO
  do i=0,360
    call f(dble(i)/360*PI, a(i))
  enddo
!$OMP END PARALLEL DO

  ...

  subroutine f(arg, ret)
    double precision :: arg, ret, noise=1.d-6
    ret = SIN(arg) + noise
    noise = -noise
    return
  end subroutine
```

6.2 使用 Monte Carlo 方法计算 π。圆心是 (0,0)，半径是 1，处在第一象限的 1/4 圆的面积是 $\pi/4$。给一个在 [0,1] × [0,1] 之间的随机数对，它出现在这个 1/4 圆的概率是 $\pi/4$，所以拥有足够的统计数据，我们能够使用所谓的 Monte Carlo 方法计算 π（参考图 6-6）。编写一个执行这个任务的并行 OpenMP 程序。为所有线程使用合适的子过程获取独立随机数序列。在弱扩展情形下，确保 162
增加更多的线程来提高统计数据。

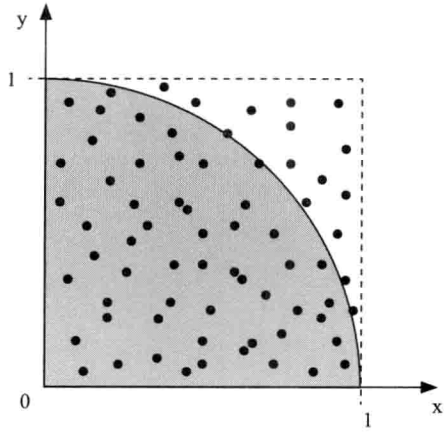


图 6-6 使用 Monte Carlo 方法计算 π（参考习题 6.2）。随机点出现在这个 1/4 圆的概率是 $\pi/4$

6.3 解开临界区。在 6.1.4 节，我们证明了使用命名临界区能够阻止死锁。如何简单修改实例代码避免使用名字？

6.4 同步风险。下面的代码有什么错误？

```
!$OMP PARALLEL DO SCHEDULE(STATIC) REDUCTION(+:sum)
  do i=1,N
    call do_some_big_stuff(i,x)
    sum = sum + x
    call write_result_to_file(omp_get_thread_num(),x)
!$OMP BARRIER
  enddo
!$OMP END PARALLEL DO
```

6.5 非平行化？（这个问题出现在 2007 年的 OpenMP 官方邮件列表中。）使用 OpenMP 并行执行下面代码中的循环。

```
double precision, parameter :: up = 1.00001d0
double precision :: Sn
```

```
3 double precision, dimension(0:len) :: opt
4
5 Sn = 1.d0
6 do n = 0,len
7   opt(n) = Sn
8   Sn = Sn * up
9 enddo
```

简单地在循环之前写上 OpenMP 工作共享指令不会有效果，这是因为代码中的循环是有依赖的：每次迭代依赖于前一次迭代的结果。并行化的代码应该独立工作于所使用的 OpenMP 调度。尽量避免使用影响串行性能的高代价操作。

解决这个问题，你可以考虑使用 OpenMP 的 FIRSTPRIVATE 和 LASTPRIVATE 语句。LASTPRIVATE 只能在工作共享循环结构体中使用，对列表中的变量值的影响是当并行循环退出时将它们最后一次循环迭代的值复制到全局变量中。

6.6 Gauss-Seidel 流水线。发明一种 Gauss-Seidel 扫描的重构方式（代码清单 6-6）以便内层循环不含有循环依赖。提示：从网格中任意选择一些位置，可视化那些能够同时更新并遵循循环依赖的其他位置。请问对基于 cache 的处理器和向量处理器，这种构造有什么性能影响（参考 1.6 节）？

163 ~ 164

高效 OpenMP 编程

OpenMP 似乎是最简单的编写并行程序的方式，这是因为它具有基于指令的简单接口和逐一处理程序中的循环而不必大量重构代码的增量式并行化。然而事实说明，除了一些显而易见的案例外，要想编写真正可扩展的 OpenMP 程序可是一件重大的任务。本章详述 OpenMP 共享内存编程中出现的性能问题以及应对措施。我们将采取第 3 章介绍的稀疏 MVM 的 OpenMP 并行化代码为例。

学术界存在着大量关于 OpenMP 程序可行优化的文献 [P12，O64]。虽然本章仅介绍最相关的基本内容，但是对于起步来说已经足够。

7.1 OpenMP 程序性能分析

在串行优化中，性能分析工具经常能够找到引起 OpenMP 性能问题的根源。最简单的例子是采取 2.1 节中介绍的对于单线程的任何一种方法，并且比较不同标量的配置文件。这个策略有几个缺点，其中最重要的是标量工具不拥有具体 OpenMP 特性的概念。在标量配置文件中，OpenMP 结构类似于组复制、同步点、自旋循环、加锁、临界区域，以及能够或多或少通过加密名字推断出被编译器打包成单独函数的部分用户代码的功能。

利用更高级的工具可以直接确定负载不均、串行分解，以及 OpenMP 循环开销等（请参考下面关于这些问题的更多讨论）。在撰写本书的时候，几乎没有免费的产品级 OpenMP 性能分析工具，并且在 OpenMP 3.0 标准中的任务介绍为工具开发者带来很多复杂的问题。

图 7-1 展现了使用 Windows 平台下的 Intel 线程性能分析工具对同一代码的不同运行事件时间轴比较 [T23]。一个事件时间轴包含应用程序相对于时间的行为信息。对于 OpenMP 性能分析，就是典型结构像并行循环、同步点、加锁，以及负载不均或者不充分并行等性能问题。作为一个简单基准，我们选择下三角矩阵和向量乘积为例：


```
1   do k=1,NITER
2 !$OMP PARALLEL DO SCHEDULE(RUNTIME)
3     do row=1,N
4       do col=1,row
5         C(row) = C(row) + A(col,row) * B(col)
6       enddo
7     enddo
8 !$OMP END PARALLEL DO
9   enddo
```

（注意这里内层循环变量的变量私有化没有必要，这是因为 Fortran 中自动实现，但是 C/C++ 不成立。）如果使用静态调度，这个问题显然遭遇严重的负载不均。图 7-1 下面的面板展示的是两个线程的 STATIC 调度时间轴，下面的线程（黑色显示）是一个用于管理目的的“看守”线程并且可以被忽略，这是因为它不执行用户代码。在遇到第一个并行区域之前，只有一个线程执行。在那之后，每一个线程使用不同颜色或者表示不同行为编码的阴影显

示。孵化阴影区表示“自旋”，也就是，这个线程在循环中等待直到被给予更多的工作或者遇到同步点（实际上，一段时间后，自旋线程常常进入睡眠状态以便释放资源；这可以在第二个同步点前观察到。这种行为往往受到用户的影响）。正如期望，静态调度导致很强的负载不均以至于第一个线程的 CPU 时间有一半被浪费。采用 STATIC，16 调度（上面的面板），这种不均消失并且性能提高了大约 30%。

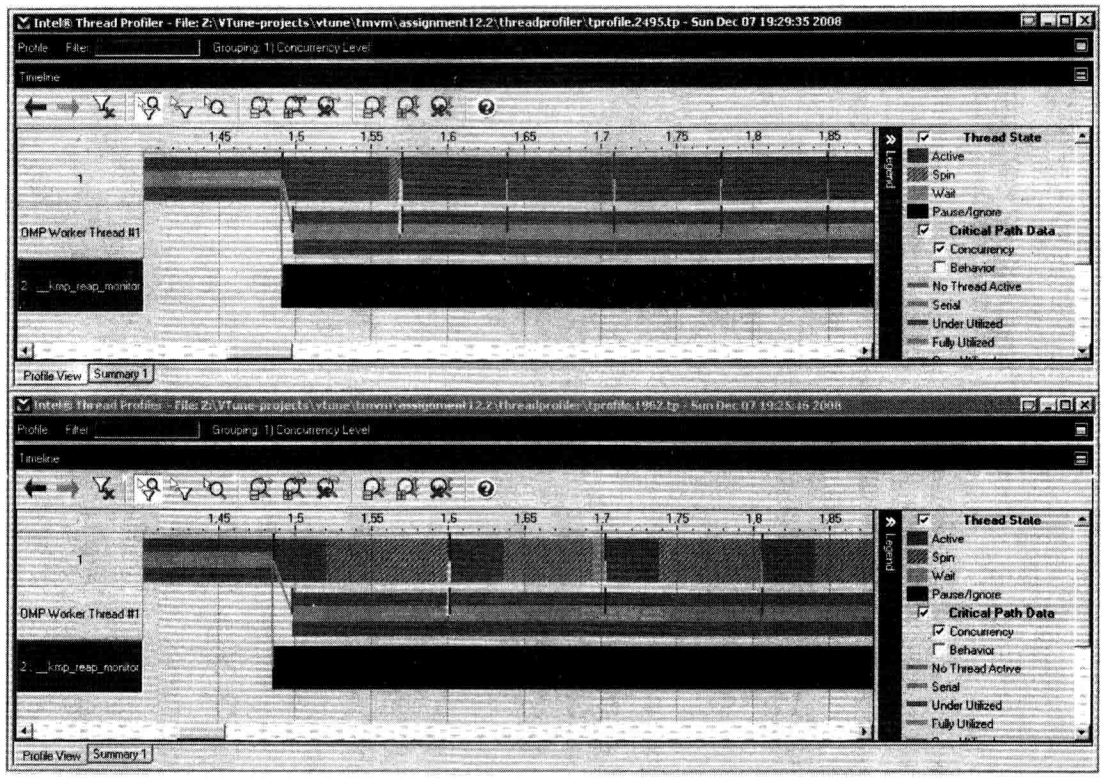


图 7-1 关于线程化代码的事件时间轴比较（三角矩阵向量乘），采用“STATIC，16”（上面的面板）和“STATIC”（下面的面板）OpenMP 调度

线程性能分析通常不仅胜任时间轴显示。通常，一个简单全局概要指示，例如，同步点、自旋循环、临界区域，或者加锁的阻塞时间比例能够揭示一些性能问题。

7.2 性能缺陷

正如其他的并行方法，OpenMP 倾向于并行编程中的标准问题：串行分解（Amdahl 法则）和负载不均，两者都在第 5 章讨论过。对于共享内存来说，通信（以数据传输的形式）通常不是什么问题，毕竟在一个计算节点中的访问延迟很小并且带宽很大（请参考第 8 章的相关问题）。负载不均问题可以选择合适的 OpenMP 调度策略来解决（请参考 6.1.6 节）。但是

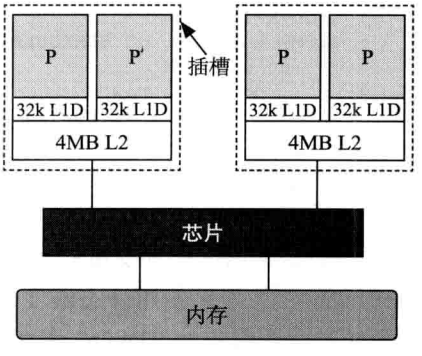


图 7-2 本章大多数基准使用的双槽双核 Xeon 5160 节点

仍然存在一般共享内存编程和 OpenMP 内在相关的特定性能问题。在这一节中我们将试着给出一些避免典型的 OpenMP 性能缺陷的实践经验。 166 ~ 167

7.2.1 减轻 OpenMP 共享区开销

不管一个并行区域开始或者终止，抑或一个并行循环启动或者结束，总存在一些不可避免的开销。线程必须复制或者至少从空闲状态唤醒，每个线程的工作包（组）的大小必须确定，在任务分配或者动态 / 指导性调度方案下，每个可使用的线程必须提供一个新的任务以继续工作，并且默认存在于工作共享结构或者并行区域的同步点负责同步所有线程。依据 5.3.6 节中讨论的改进后可扩展模型，这些贡献被看作“通信开销”。因为它们倾向线性于线程数量，所以 OpenMP 似乎不太适合很强的扩展情形；假设 N 个线程，加速比是

$$S_{\text{omp}}(N)=\frac{1}{s+(1-s)/N+\kappa N+\lambda},\qquad (7\text{-}1)$$

其中 λ 表示 N 个无关开销。当 N 很大时，这个表达式趋于零，似乎 OpenMP 编程模型已经够好了。然而实践中，一切没有损失：当然性能作用依赖于 κ 和 λ。如果遵循一些简单的指导建议，OpenMP 开销的负作用则可以很大程度减少。

1. 如果并行不够好，那么使用串行代码

这可能是 OpenMP 最常见的性能问题。如果工作共享结构每个线程不能包含足够“工作”，例如一个简单循环的每次迭代执行时间很少，OpenMP 开销将会导致很差的性能。如果循环次数少于一定阈值，那么串行版程序更是一个好的选择。OpenMP 的 IF 语句有助于这种情形：

```
1 !$OMP PARALLEL DO IF(N>1700)
2   do i=1,N
3     A(i) = B(i) + C(i) * D(i)
4   enddo
5 !$OMP END PARALLEL DO
```

168

图 7-3 显示的是在双槽 Xenon 5160 节点上（概略图如图 7-2）向量三联数据纯串行，OpenMP 上的一个和四个线程的比较。即使使用单线程，在很小的 N 时，OpenMP 也会带来开销（请参考下面关于工作共享结构开销的讨论）。如果阈值选择合理，那么使用 IF 语句会产生一个优化的线程和串行循环版本的组合，并且当超大循环长度不能保证时，这是强制的。然而，N 小于 1000 时，将存在一些可测量的性能点；毕竟，相对于串行情形，更多的代码将被执行。注意 IF 语句能够（部分）解决这种问题，但是不是并行循环开销的原因。接下来将详尽地介绍可以减少它的方法。

避免取消所有并行执行，使用 NUM_THREADS 语句也可能是一种减少特定并行区域线程数量的方式：

```
1 !$OMP PARALLEL DO NUM_THREADS(2)
2   do i=1,N
3     A(i) = B(i) + C(i) * D(i)
4   enddo
5 !$OMP END PARALLEL DO
```

越少的线程就意味越少的开销，因此最终的性能将好于使用 IF，至少对于某些循环长度如此。 169

2. 避免隐式同步点

需要意识大多数的 OpenMP 工作共享结构（包括 OMP DO/END DO）会自动在结构结

束时插入同步点。这种默认行为是安全的，因为这样能够保证所有的线程能够在结构体结束前完成各自分担的工作。在不需要同步点的情况下，可以使用 NOWAIT 语句移除隐式同步点：

```
!$OMP PARALLEL
!$OMP DO
  do i=1,N
    A(i) = func1(B(i))
  enddo
!$OMP END DO NOWAIT
! still in parallel region here. do more work:
!$OMP CRITICAL
  CNT = CNT + 1
!$OMP END CRITICAL
!$OMP END PARALLEL
```

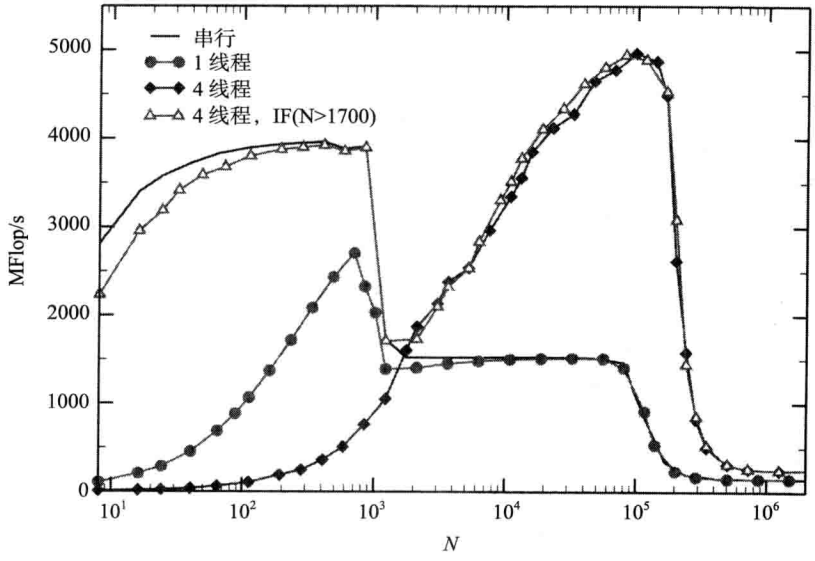


图 7-3 OpenMP 开销和 IF（N>1700）语句效益对于向量三联基准。（如图 7-2 Intel Xeon 5160 3.0GHz 双槽双核系统，Intel 10.1 编译器）

在并行区域末尾的隐式同步点是不能够移除的。虽然类似临界区域这样的隐式同步点会带来同步开销，但是经常需要它们来避免竞争条件。程序员需要细心检查 NOWAIT 语句是否安全。

下面的 7.2.2 节显示了实验上如何确定工作共享循环标准实例的同步开销。

3. 尽量减少并行区域数量

一般我们会尽最大可能并行化嵌套循环，并且和前面的指导性建议保持内在一致性。但是并行化内层循环会导致 OpenMP 开销的增长，这是因为将创建或者唤醒一组线程：

```
  double precision :: R
  R = 0.d0
  do j=1,N
!$OMP PARALLEL DO REDUCTION(+:R)
    do i=1,N
      R = R + A(i,j) * B(i)
    enddo
!$OMP END PARALLEL DO
```

```
9       C(j) = C(j) + R
10    enddo
```

在这个特殊例子中，一组线程将重新启动 N 次，j 循环的每次迭代重启一次。将并行结构移至外层循环将减少 1 个线程的重启，并且获取额外的好处，这是因为所有线程将工作在结果向量的不同部分，所以 reduction 语句可以省略：

```
1  !$OMP PARALLEL DO
2    do j=1,N
3      do i=1,N                                          170
4        C(j) = C(j) + A(i,j) * B(i)
5      enddo
6    enddo
7  !$OMP END PARALLEL DO
```

线程组创建或者重启越少，引入开销越少。这种策略需要额外的编程技巧，这是因为如果线程组在共享工作结构间继续运行，那些被单个线程执行的代码将会冗余地被其他所有线程执行。思考下面的例子：

```
1    double precision :: R,S
2    R = 0.d0
3  !$OMP PARALLEL DO REDUCTION(+:R)
4    do i=1,N
5      A(i) = DO_WORK(B(i))
6      R = R + B(i)
7    enddo
8  !$OMP END PARALLEL DO
9    S = SIN(R)
10 !$OMP PARALLEL DO
11   do i=1,N
12     A(i) = A(i) + S
13   enddo
14 !$OMP END PARALLEL DO
```

循环之间的 SIN 函数仅被主线程调用。在第一次循环结尾，所有的线程同步并且甚至有可能进入休眠，它们在第二次循环执行时唤醒。为了避免这种开销，可以采用连续的并行区域：

```
1    double precision :: R,S
2    R = 0.d0
3  !$OMP PARALLEL PRIVATE(S)
4  !$OMP DO REDUCTION(+:R)
5    do i=1,N
6      A(i) = DO_WORK(B(i))
7      R = R + B(i)
8    enddo
9  !$OMP END DO
10   S = SIN(R)
11 !$OMP DO
12   do i=1,N
13     A(i) = A(i) + S
14   enddo
15 !$OMP END DO NOWAIT
16 !$OMP END PARALLEL
```

现在的第 10 行的 SIN 函数将由所有的线程计算，并且 S 必须是私有化的。第 2 次循环可以安全使用 NOWAIT 语句来减少同步开销。对于第一次循环来说这是行不通的，因为最终的计算结果只有在所有线程同步之后合法。 171

4. 避免“普通”负载不均

任务数量或者并行循环跳脱计数，相对于线程数量应该很大。如果跳脱计数是很小的、

不是线程数量整数倍数，那么一些线程相对于其他线程做很少的工作，从而导致负载不均。这种作用独立于其他的负载均衡问题或者开销问题，也就是，当每个任务的工作量相近并且OpenMP开销可以忽略时，它仍然会发生。

一个典型情况就是当在高线程化架构上运行深层次嵌套循环时，这种情况可能很重要[O65]（更多关于硬件线程化信息请参考1.5节）。线程数量越多，并行（外层）循环上每个线程获取的任务将会越少：

```
1   double precision, dimension(N,N,N,M) :: A
2 !$OMP PARALLEL DO SCHEDULE(STATIC) REDUCTION(+:res)
3   do l=1,M
4     do k=1,N
5       do j=1,N
6         do i=1,N
7           res = res + A(i,j,k,l)
8         enddo ; enddo ; enddo ; enddo
9 !$OMP END PARALLEL DO
```

这里的外层循环是并行化显然的候选，因为这样导致被执行工作共享循环（和隐式同步点）的数量最少并生成最少开销。然而，外层循环长度M可能十分小。在最佳可能条件下，如果t表示线程数量，则一次迭代中会有t-mod(M, t)个线程获得块组小于其他的线程。如果M/t很小，负载不均将影响扩展性。

COLLAPSE语句（OpenMP 3.0引入）有助于解决这种情况。对于完美循环嵌套，也就是，在不同的do（和enddo）语句之间没有代码并且循环次数不相互依赖，这条语句将一定数量的循环层次合一。自动计算原始的循环索引以便循环体保持不变：

```
1   double precision, dimension(N,N,N,M) :: A
2 !$OMP PARALLEL DO SCHEDULE(STATIC) REDUCTION(+:res) COLLAPSE(2)
3   do l=1,M
4     do k=1,N
5       do j=1,N
6         do i=1,N
7           res = res + A(i,j,k,l)
8   enddo ; enddo ; enddo ; enddo
9 !$OMP END PARALLEL DO
```

这里最外层循环合成的循环长度为M×N，生成的长循环将被所有的线程并行执行。这
172 将改善负载均衡问题。

5. 避免动态/指导性循环调度或任务分配

所有并行循环调度选项（除了STATIC）和任务结构需要一些重要计算或者记录以便能够找出那个用来计算下一个块组或者任务的线程。如果每个任务包含很少的工作，那么开销会非常小。通过一个简单基准测试，可以粗略估计为一个线程分配新的循环块组的开销。图7-4显示了对于一个简单的纯计算负载并行循环的静态和动态调度的性能比较：

```
1 !$OMP PARALLEL DO SCHEDULE(RUNTIME) REDUCTION(+:s)
2   do i=1,N
3     s = s + compute(i)
4   enddo
5 !$OMP END PARALLEL DO
```

compute()函数执行一些寄存器内部计算（例如，超越函数估值）将花费几百个周期。
173 它具体做什么并不重要，只要它不干扰主并行循环也不引起存储级别的带宽瓶颈。N应该足够大以便OpenMP循环的启动开销可以忽略不计。t个线程的性能基线——$P_s(t)$——是以每秒百万迭代次数为单位，测量时不包含块组的静态调度（请参考图7-4中运行在图7-2描绘

的 Xeon 5160 双核双槽节点上两个核中的两个线程）。这个基线不依赖线程和核的绑定（在 cache 组内部、跨越不同的 ccNUMA 领域等）。这是因为每一个线程仅执行一个块组，并且任何的 OpenMP 开销都只发生在工作共享结构的开始和结尾。当 N 很大时，静态基线将比纯串行的性能大 t 倍。

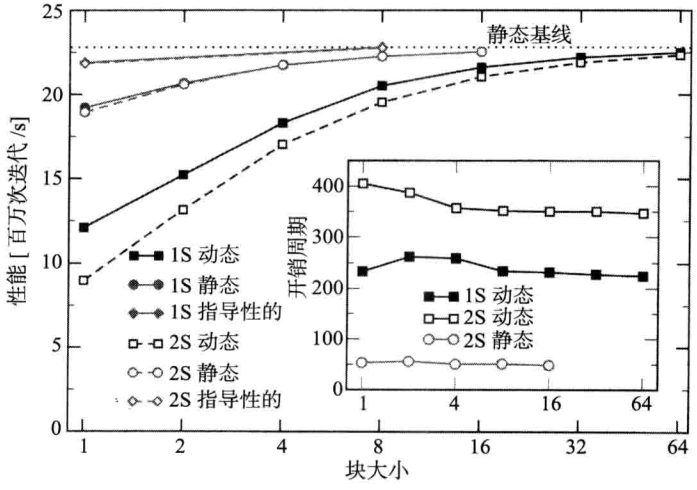


图 7-4 主面板：运行在类似图 7-2 所示的 Intel Xeon 5160 3.0 GHz 双槽双核系统上的普通大循环次数的工作共享循环，分别在 L2 组中的双核（1S）或者不同槽（2S）静态（满周期）和动态（开放符号）调度性能对比（Intel 10.1 编译器）。插图：以分配给每个线程单个块组的处理器周期计算开销

不管何时块组被任何调度变体使用而为一个线程分配新块组花费的时间被看做开销。图 7-4 中的主面板显示了当线程分别运行在一个 L2cache 组（关闭符号）和不同槽（开放符号）上的静态（圆圈）、动态（方块）和指导性（菱形）调度性能数据。正如所期望，小块组的开销最大，并且仅在动态调度占主要地位。指导性调度表现最好，这是因为在循环开始时大块组被分配，并且显示的块组还是最小的边界（请参考图 6-2）。槽间和 L2 内情形的区别仅对于动态调度来说很明显，这是因为任何工作分布都需要一些公共资源。如果公共资源保持在共享 cache 中，那么块组分配会更快。尽管要快于指导性调度，但是由于平均块组值很大，所以效果不明显。

如果 $P(t, c)$ 是在块组大小为 c 的 t 个线程的性能，那么每次迭代执行的每个线程的静态基线和“分块”版本的区别是每次迭代开销。每个完整块组是

$$T_o(t, c) = \frac{t}{c}\left(\frac{1}{P(t, c)} - \frac{1}{P_s(t)}\right) \qquad (7\text{-}2)$$

图 7-4 的插图显示以 CPU 周期计算的开销基本上与至少大于 4 块的块组大小无关。当两个线程都运行在 L2 组里，给一个线程分配新的块组将花费 200 周期，然而运行在不同槽上则花费 350 周期（这些时间包括静态调度中每个块组 50 个周期的开销）。我们又遇到了决定性的具有互斥线程安排或者与其相近的情形。

请注意，尽管一般性方法论适合于所有的共享内存系统，但是分析结果还是依赖于硬件特性和编译器对 OpenMP 工作共享结构的具体实现。实际的数量可能不同的平台差别很大。

其他在这个基准中没有提到的因素能够影响动态调度的工作共享结构的性能。在内存约

束的循环代码中，如果块组很小，则每次预取不高效或者不可能实现。进一步，由于内存访问的随机性，ccNUMA 局部性很难保持，这也更倾向于指导性调度。关于这个问题的详细
174 细节请参考 8.3.1 节。

7.2.2 决定短循环的 OpenMP 开销

这个问题产生于如何估计一个并行工作共享结构可能带来的开销。一般地，开销由两部分组成：固定部分和依赖于线程数量的变动部分。虽然不同系统的开销存在巨大差异，但是通常是 CPU 周期几百倍或者几千倍数量级。很容易通过底层基准获取数据的简单性能模型来计算开销。作为一个例子，我们选取短长度向量三元组并且采取静态调度以便当每个核拥有自己的 L1 时能够并行运行不同量级（由于共享 cache 或者主存通常是瓶颈，尤其在多核系统中，所以大向量性能将得不到扩展）：

```
1 !$OMP PARALLEL PRIVATE(j)
2   do j=1,NITER
3 !$OMP DO SCHEDULE(static) NOWAIT   ! nowait is optional (see text)
4     do i=1,N
5       A(i) = B(i) + C(i) * D(i)
6     enddo
7 !$OMP END DO
8   enddo
9 !$OMP END PARALLEL
```

通常，选择 NITER 以便全部时间能够被准确计算并且一次启动（第一次加载数据到 cache）的影响变得不再重要。NOWAIT 语句是可选的，这里仅用来证明在循环工作共享结构中的隐式同步点的影响（参考下面）。

假设基于 t 个线程的问题规模为 N 的性能模型的全部运行时间被分成计算和建立或者关闭部分：

$$T(N, t) = T_c(N, t) + T_s(t) \tag{7-3}$$

进一步假设我们已经衡量了纯串行性能 $P_s(N)$，这样我们可以写成

$$T_c(N, t) = \frac{2N}{tP_s(N/t)} \tag{7-4}$$

这允许与 OpenMP 开销无关的 N 个独立性能行为。分子中的因子 2 是因为性能是以 MFlop/s 计算的并且每次循环迭代含有两个 Flop。如上所述，建立或者关闭时间由固定延迟部分和依赖于线程数量的部分组成：

$$T_s(t) = T_1 + T_p(t) \tag{7-5}$$

现在我们可以计算在问题规模为 N，线程数为 t 的并行性能：

175

$$P(N, t) = \frac{2N}{T(N, t)} = \frac{2N}{2N[tP_s(N/t)]^{-1} + T_s(t)} \tag{7-6}$$

图 7-5 展示了在四个核（两个槽）上 N 很小向量三元组的性能数据，包括对模型（见式（7-6））上单线程（圆圈）和四线程（方块）情形参数适应，并且通过 nowait 语句消除隐式同步点（开放符号）。这表明以纳秒计算的合适参数用 $T_s(t)$，正如式（7-5）中定义。用来计算串行版本 P_s 值的数据并不能直接使用需要如下进行近似

$$P_s(N) = \frac{2N}{2N/P_{max} + T_0} \tag{7-7}$$

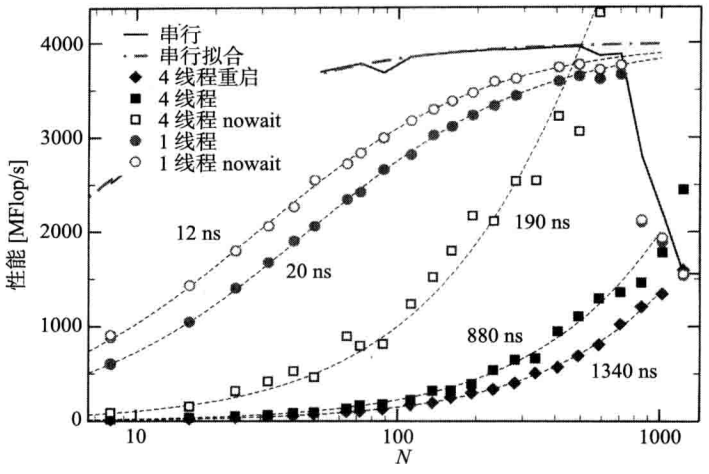


图 7-5 N很小的向量的 OpenMP 循环开销 $T_s(t)$ 是由式（7-6）模型确定以用来测量性能数据的。注意隐式同步点的影响（被 nowait 语句移除）。方块显示的数据是从每次循环启动并行区域获得的。(Intel Xeon 5160 3.0 GHz 双核双槽节点（参考图 7-2，Intel 编译器 10.1))

其中 P_{max} 是串行性能渐近值，并且 T_0 综合了所有的标量开销（流水线启动、循环分支错误预测等）。所有参数都是为衡量串行性能数据而决定的（图 7-5 中点虚线）。然后，将式（7-7）代入式（7-6）：

$$P(N,t)=\frac{1}{(tP_{max})^{-1}+(T_0+T_s(t))/2N} \tag{7-8}$$

惊奇的是，这里运行单独的 OpenMP 线程相比于纯串行代码的开销是可衡量的。然而，当 $N\lesssim 1000$ 时，这两个版本达到近似的性能，不能把这种效果归因于不高效的标量代码，尽管 OpenMP 有时阻止高级别循环优化。单线程开销起源于编译器无法实现，只好生成一个串行执行的单独代码版本。

在单线程同步点可以忽略，并且占据四个线程开销的绝大部分。但是尽管不考虑它，在那样的案例中建立工作共享循环也花费大约 190ns（570 个 CPU 周期）(假设同步点是通过一个必须被所有线程顺序更新的内存变量实现，那么确实可以估计同步点开销的平均内存延迟)。被标记“重启”的数据（方块）是通过使用一个组合的 parallel do 指令获取的，这样使得一组线程在内层循环执行时被唤醒：

```
1    do j=1,NITER
2  !$OMP PARALLEL DO SCHEDULE(static)
3      do i=1,N
4        A(i) = B(i) + C(i) * D(i)
5      enddo
6  !$OMP END PARALLEL DO
7    enddo
```

将唤醒线程时间从同步点和工作共享开销分离出来是可能的。这里有近 460ns（1380 个时钟周期）用来重启线程组的额外开销，这也证明了上面提到的，最小化 OpenMP 程序中的并行区域数量是有好处的。

类似于前面章节中的动态调度试验，小 OpenMP 循环引起的实际开销依赖像编译器、运行时库、cache 组织和一般节点架构等许多因素影响。线程组是在一个高速缓冲组还是在几 176

个高速缓冲组中有很大差别 [M41]（如何使用姻亲机制信息请参考附录 A）。一般性结果是隐式（还有显式）同步点占据 OpenMP 程序开销很大一部分。EPCC OpenMP 微基准包 [136] 提供了测量 OpenMP 相关参数的框架，包括同步点，但是使用的方法不同于这里介绍的。

7.2.3 串行化

协调共享资源访问的最直接方式是使用临界区域。除了注意一些事宜，临界区域还存在着串行执行代码的可能性。在如下例子中，矩阵 M 的列被并行更新。col_calc() 函数决定哪一列被更新：

```
1   double precision, dimension(N,N) :: M
2   integer :: c
3   ...
4  !$OMP PARALLEL DO PRIVATE(c)
5   do i=1,K
6     c = col_calc(i)
7  !$OMP CRITICAL
8     M(:,c) = M(:,c) + f(c)
9  !$OMP END CRITICAL
10  enddo
11 !$OMP END PARALLEL DO
```

177

函数 f() 返回一个用来累加矩阵 c 列的数组。既然每一列可能被更新多余一次，那么数组和应该被临界区域保护。但是，如果第 8 行花费了 CPU 大部分时间，那么程序串行执行能获得很好的效率并且使用多于一个线程没有什么好处；因为进入临界区域需要额外开销，所以这个程序运行较慢。

粗粒度保护资源（这里指的是矩阵 M）的方式通常称为 *big fat lock*。一种可以使用资源子结构的情形，也就是，矩阵是由独立的列组成，并且每一列独立访问。串行化仅当两个线程都试图更新同一列时才发生。遗憾的是，由于名字不能当做变量，因此命名临界区域（请参考 6.1.4 节）没有什么用处。然而，可以为每一列使用单独的 OpenMP 锁变量：

```
1   double precision, dimension(N,N) :: M
2   integer (kind=omp_lock_kind), dimension(N) :: locks
3   integer :: c
4  !$OMP PARALLEL
5  !$OMP DO
6   do i=1,N
7     call omp_init_lock(locks(i))
8   enddo
9  !$OMP END DO
10  ...
11 !$OMP DO PRIVATE(c)
12  do i=1,K
13    c = col_calc(i)
14    call omp_set_lock(locks(c))
15    M(:,c) = M(:,c) + f(c)
16    call omp_unset_lock(locks(c))
17  enddo
18 !$OMP END DO
19 !$OMP END PARALLEL
```

如果被 col_calc() 函数控制的 i 到 c 的映射不只是几列更新，那么通过这种优化方式并行化能得到很强的加强（请参考 [A83] 的真实例子）。需要意识到设置和去设置 OpenMP 锁会引起一些开销，正如进入和离开临界区域。如果这种开销与更新矩阵一行开销相比，则细粒度同步方案没有什么益处。

如果内存是稀缺资源，那么存在一种解决方案：可以使用共享数据线程局部拷贝将其“拉在一起”，举例来说通过在并行区域的结尾进行归约操作。在我们的例子中，通过在 M 上进行 OpenMP 归约很容易实现它。如果 K 足够大，那么额外开销可以忽略： 178

```
   double precision, dimension(1:N,1,N) :: M
   integer :: c
   ...
!$OMP PARALLEL DO PRIVATE(c) REDUCTION(+:M)
  do i=1,K
    c = col_calc(i)
    M(:,c) = M(:,c) + f(c)
  enddo
!$OMP END PARALLEL DO
```

注意数组上的归约仅在 Fortran 中允许，并且有进一步限制 [P11]。在 C/C++ 中，必须在并行区域中执行显式私有化（可能使用堆内存）并且人为使用归约，正如列表 6-2 所示。

7.2.4 伪共享

在 4.2.1 节中介绍的基于硬件一致性的 cache 在共享内存系统中利用 cache，这对于程序员来说是透明的。然而，在一些例子中，一致性 cache 通信量可能将性能压制到非常低的水平。这种情形可能发生，如果相同 cache 行被一组线程连续更改以便一致性 cache 逻辑被强制驱除并且迅速连续地重新加载。作为一个例子，考虑使用维度为 {1,⋯,8} 的数组 A 的一些大数计算直方图的程序片段：

```
integer, dimension(8) :: S
integer :: IND
S = 0
do i=1,N
  IND = A(i)
  S(IND) = S(IND) + 1
enddo
```

在一个直接并行化中，很可能使 S 成为两维，这样可以为每个线程的局部直方图保存空间以避免数组 S 上的共享资源串行化：

```
   integer, dimension(:,:), allocatable :: S
   integer :: IND,ID,NT
!$OMP PARALLEL PRIVATE(ID,IND)
!$OMP SINGLE
   NT = omp_get_num_threads()
   allocate(S(0:NT,8))
   S = 0
!$OMP END SINGLE
   ID = omp_get_thread_num() + 1
!$OMP DO
   do i=1,N                                                    179
     IND = A(i)
     S(ID,IND) = S(ID,IND) + 1
   enddo
!$OMP END DO NOWAIT
   ! calculate complete histogram
!$OMP CRITICAL
   do j=1,8
     S(0,j) = S(0,j) + S(ID,j)
   enddo
!$OMP END CRITICAL
!$OMP END PARALLEL
```

在第 18 行开始的循环收集所有线程的部分结果。尽管这是一个合法的 OpenMP 程序，但是它使用四个线程并不一定比使用一个线程快。理由是两维数组 S 包含来自所有线程的直方图数据。对应于多数处理器上的两个或者三个 cache 行，使用四个线程时存在 160 字节。在第 13 行中针对 S 更改的每个直方图执行写的 CPU 都必须获取任一高速缓冲行的所有权；几乎每次写都导致 cache 失效并且引起后续一致性阻塞，这可能是因为在修改模式下需要高速缓冲行处在另一个处理器的 cache 中。相对于 S 适合单 CPUcache 的串行代码中，这样会带来很不好的性能。

在简单例子中，程序员能够通过标准的编译的寄存器优化消除错误共享。如果关键的更新操作能够作用于内容仅在循环末尾写出的寄存器，这样将没有 cache 失效出现。在上面的例子中这是不可能的，这是因为第 13 行计算 S 的第二个指示变量。

一旦发现问题，通过手动优化来避免错误共享其实是一个简单任务。一个标准技术就是数组补全（array padding），也就是，不同线程更新内存位置之间插入合适的空间。在上述直方图例子中，第 6 行 S(0:NT*CL,8) 语句分配 S，其中 CL 是整数大小的 cache 行，这将为每个线程保留一个独占 cache 行。当然，在程序其他地方，S 的第一个指示变量不得不乘以 CL（转置 S 将会省些内存，但是主要问题依旧存在）。

一个很不错的解决方案以数据私有化形式存在（请参考上面的 7.2.3 节）：在并行区域的入口处，每个线程将在它们自己的栈空间中获取直方图数组的一份本地副本。不同实例占据相同的高速缓冲行是不可能的，所以错误共享不是一个问题。进一步，使用 REDUCTION 语句可以获得和前面纯串行代码的一个简化等价版本：

```
1   integer, dimension(8) :: S
2   integer :: IND
3   S=0
4  !$OMP PARALLEL DO PRIVATE(IND) REDUCTION(+:S)
5   do i=1,N
6     IND = A(i)
180 7     S(IND) = S(IND) + 1
8   enddo
9  !$OMP EMD PARALLEL DO
```

如果代码不采用 OpenMP 编译，则需要设置 S 为零。正如 reduction 语句，OpenMP 支持使用合适起始值来自动初始化变量。

重申一下，因为我们使用 Fortran，所以可以使用最方便的形式（归约）私有化（OpenMP 标准不允许 C/C++ 对数组使用归约）并且 REDUCTION 语句支持基本的运算操作（加法）。然而，即使不使用归约语句，所需的运算也很容易显式构造（参见习题 7.1）。

7.3 案例分析：并行稀疏矩阵向量乘

作为 OpenMP 对重要问题的有趣应用，现在我们考虑稀疏 MSM 数据布局并且通过并行化 3.6 节中的 CRS 和 JDS 矩阵向量乘代码来优化 [A84, A82]。

无论选择两种存储格式中的哪种，一般的并行方法都是一样的：所有实例都有计算结果向量的连续元素（或者块元素）的可并行化循环（见图 7-6）。对于 CRS 矩阵格式，可以直接采用这个原则。

```
1  !$OMP PARALLEL DO PRIVATE(j)⊖
181 2    do i = 1,Nr
```

⊖ 在外层循环的词法范围中私有化内层循环的指示变量在 Fortran 不是必需的，但是在 C/C++ 是必需的。

```
3       do j = row_ptr(i), row_ptr(i+1) - 1
4         C(i) = C(i) + val(j) * B(col_idx(j))
5       enddo
6     enddo
7   !$OMP END PARALLEL DO
```

鉴于外层循环很长，这里 OpenMP 开销不是问题。然而由于依赖于具体的矩阵形式，如果非常短的或者很长的矩阵行在某些区域聚集，那么一些负载不均可能会出现。像 DYNAMIC 或者 GUIDED 的不同 OpenMP 调度策略可能有助这样的情形。

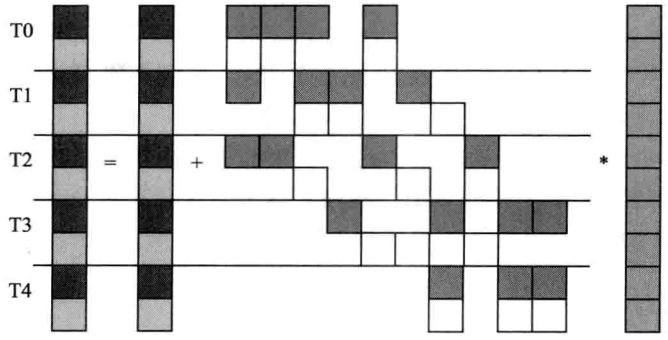


图 7-6 稀疏 MVM 的并行化方法（五个线程）。所有标记的元素都在并行化循环的一个迭代中处理。RHS 向量被所有线程访问

JDS 的 MVM 的并行化也很简单：

```
1   !$OMP PARALLEL PRIVATE(diag,diagLen,offset)
2     do diag=1, Nj
3       diagLen = jd_ptr(diag+1) - jd_ptr(diag)
4       offset = jd_ptr(diag) - 1
5   !$OMP DO
6       do i=1, diagLen
7         C(i) = C(i) + val(offset+i) * B(col_idx(offset+i))
8       enddo
9   !$OMP END DO
10    enddo
11  !$OMP END PARALLEL
```

在这个例子中并行循环是内存循环，但是由于循环次数很大，所以这里并不存在 OpenMP 开销问题。进一步，相对于 CRS 并行版本，由于所有内层迭代包含等量工作量，所以也不存在负载不均问题。所有一切看起来很完美，但是对于 JDS 的 MVM 来说并不是这样。然而，循环展开和分块版本应该能够很好等价地并行化。对于块化代码（见图 3-19），所有块上的外层循环是很自然的候选：

```
1   !$OMP PARALLEL DO PRIVATE(block_start,block_end,i,diag,
2   !$OMP& diagLen,offset)
3     do ib=1,Nr,b
4       block_start = ib
5       block_end   = min(ib+b-1,Nr)
6       do diag=1,Nj
7         diagLen = jd_ptr(diag+1)-jd_ptr(diag)
8         offset  = jd_ptr(diag) - 1
9         if(diagLen .ge. block_start) then
10          do i=block_start, min(block_end,diagLen)
11            C(i) = C(i)+val(offset+i)*B(col_idx(offset+i))
12          enddo
13        endif
```

```
14        enddo
15      enddo
16  !$OMP END PARALLEL DO
```

这个版本甚至能够获得更少的 OpenMP 开销，这是因为 DO 指令在最外层。遗憾
182 的是，由于矩阵行按照大小排序，所以仍然存在负载不均的可能。但是由于依赖于循环指示变量的负载可以粗略估计，所以具有一个线程大小的块组静态调度可以避免这种影响。

图 7-7 展现的是在三种不同体系结构上并行的 CRS 和块化 JDS 版本的性能和扩展行为：两个 ccNUMA 系统（Opteron 和 SGI Altix，等价于图 4-5 和图 4-6 中的块状图）和一个 UMA 系统（类似图 4-4 中的 Xeon/Core2 节点）。在所有实例中，代码尽量运行在很少的局部域或者槽上，也就是，在下一次迭代之前首先填满一个 LD 或者槽。插图显示了 LD 内或者槽内相对于单核的扩展基线。这个层次上所有系统都考虑带宽的限制。使用第二线程获取的性能远不是所期望的 2 倍。然而，这种行为还很严重依赖于单核利用局部内存能力的接口：在 Atlix 上相对低的单线程 CRS 性能导致双线程近似 1.8 倍的加速（参见 5.3.8 节）。

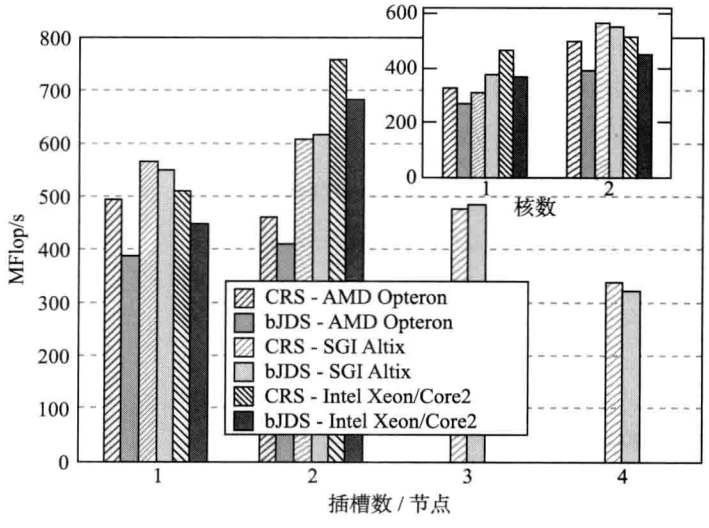


图 7-7 在三种不同体系结构上，相对于 CRS（填充条）和块化 JDS（实线条），稀疏 MVM 的 OpenMP 直接并行化的性能和较强的扩展性。Intel Xeon/Core2 系统是 UMA 类型，另外两种系统是 ccNUMA。不同的扩展基线已经被分离开（主框架中是单槽或者 LD，插图中是单核）

跨槽或者 LD 的扩展性（图 7-7 中的主框架）揭示了 ccNUMA 和 UMA 系统上的一个重要区别。当使用第二个槽时，仅 UMA 节点表现了期望的加速比，这是由于第二个前端
183 总线提供了额外的带宽（对于基于 FSB 的设计，存在跨槽带宽扩展性少于理想值的问题，所以这里我们看不到 2 倍的加速化）。尽管 ccNUMA 体系结构能够实现扩展的带宽，但是代码并不适用于 ccNUMA，表现出差的扩展性或者在 Altix 上，当 LD 数量变大时性能甚至出现下降。

ccNUMA 实现理想带宽失败的原因在于我们忽视了一个还没有考虑的扩展性的必要前

提：正确的局部访问数据和线程布局。请参考第 8 章解决这个问题的编程技巧以及 [O66] 获取现代共享内存系统上一个更一般的并行稀疏 MVM 优化评估。

习题

7.1 *私有化训练*：7.2.4 节中我们已经优化了消除错误共享的并行计算直方图的代码。最终代码采用 REDUCTION 语句来计算所有 S( ) 部分结果的和。C/C++ 中如何实现？

7.2 *超线性加速*：问题规模固定，当线程数量增加（强扩展），为应用程序获取额外的 cache 空间，存在使整个工作集进入所有使用核聚集的 cache 中。加速比将比额外数量的核能够增加的要多。你能够从 2 维并行 Jacobi 解决方法的性能数据中识别这个情形（见图 6-3）？当然，结果可能仅对于一种类型的机器合理。对于一般基于 cache 的机器需要什么条件才能使代码获取超线性加速比的可能性？

7.3 *归约和初始值*：在 7.2.1 节一些减少并行区域数量的例子中，尽管 OpenMP 自动初始化这样的变量，但是在进入并行区域前，我们显式设置归约变量 R 为 0。为什么需要这样做？

7.4 *最优线程数*：在拥有六个核、两个槽和三个核 L3 组的 ccNUMA 系统上，对于内存约束多线程应用程序，最优的线程数量是多少？ 184

ccNUMA 体系结构的局部性优化

前面关于 ccNUMA 架构小节中已经提到，对于性能受制于内存带宽、局部性和冲突问题（请参考图 8-1 和图 8-2）的应用程序，当线程或者进程以及它们的数据没有小心布局在 ccNUMA 系统的局部区域时，性能会变差。遗憾的是，现在的 OpenMP 标准（3.0）没有提到页布局，所以这将取决于程序员使用的系统构建工具。本章探讨一般的，即正确布局数据的绝大多数系统无关选项以及所能避免的缺陷。我们将说明面页面布局对于共享内存并行编程并不是什么问题。

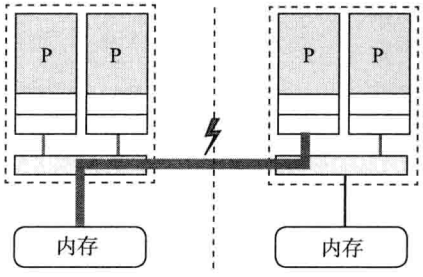


图 8-1　ccNUMA 系统上的局部性问题。被映射到局部区域的内存页被不相连的处理器访问导致 NUMA 阻塞

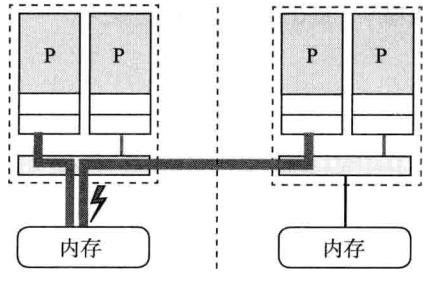


图 8-2　ccNUMA 系统上的冲突问题。尽管网络很快，但是单一局部区域并不能满足局部和非局部并发访问的带宽需求

8.1　ccNUMA 的局部访问

尽管如今 ccNUMA 架构广泛存在，但是并不是所有应用领域都拥有 ccNUMA 相关知识；内存约束性代码必须被设计为采用合适的页布局 [O67]。布局问题涉及两个方面：首先，程序员必须保证内存被映射到实际访问它们的处理器局部区域。这能最小化网络上的 NUMA 阻塞。这里的“映射”意味着用来描述虚实内存关系的页表项被设置。从而，在 ccNUMA 系统上的局部访问总是 OS 页面层次上，典型的页大小是（常见）4kB 或者（不常见）16kB，有时甚至更大。然而，当仅包含几个页的工作集的严格局部性可能很难实现，尽管此情形的问题是受 cache 限制的。其次，线程或

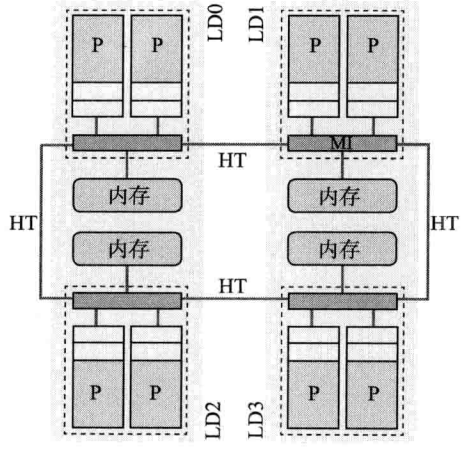


图 8-3　四个局部区域 LD0…LD3 和每个 LD 两个核的 ccNUMA 系统（基于 AMD　Opteron 双核处理器），通过高传输率网络相连。这里存在三个层次的 NUMA 访问（局部区域、一级跳跃、两级跳跃）

者进程必须被固定在最初映射它们内存区域的 CPU 上以便不失去访问局部性。下面我们假设采用了合适的关联机制（参考附录 A）。

一个含有四个局部区域的典型 ccNUMA 节点由图 8-3 给出。每个槽使用两个高速传输（HT）链接相邻区域，这样就形成了一个“封闭链式”拓扑结构。这样根据访问页面所需的 HT 跳跃（0，1，2），内存访问被分为三个层次。不同系统上拥有的不同远程带宽和延迟开销；向量测量至少能够提供粗略的指导。请参考下面关于如何控制页布局细节的内容。

注意尽管相对于局部访问，NUMA 间链接很快，但是冲突问题是不可避免的。不论链接多么快，没有 NUMA 间链接，也无法将 ccNUMA 变成 UMA。

8.1.1 首次访问方式分配页面

幸好在现在所有 ccNUMA 架构上初始化映射需求能够以一种方便的方式完成。如果配置正确（包括类似于固态软件 [“BIOS”]、操作系统和运行时库等），那么它们支持内存页的首次接触策略：当处理器第一次访问页面时，被映射到这个处理器的局部区域。仅分配内 186
存是不够的。因此在 ccNUMA 上需要关注数据初始化代码（在 C 中使用 calloc() 很可能等效）。作为一个例子，我们再次查看代码清单 1-1 中用并行 OpenMP 简单实现的向量代码。虽然不在栈上分配数组，但是我们使用动态内存（堆），稍后解释原因（为了简洁我们省略计时功能）：

```
1   double precision, allocatable, dimension(:) :: A, B, C, D
2   allocate(A(N), B(N), C(N), D(N))
3  ! initialization
4   do i=1,N
5     B(i) = i; C(i) = mod(i,5); D(i) = mod(i,10)
6   enddo
7   ...
8   do j=1,R
9  !$OMP PARALLEL DO
10    do i=1,N
11      A(i) = B(i) + C(i) * D(i)
12    enddo
13 !$OMP END PARALLEL DO
14    call dummy(A,B,C,D)
15  enddo
```

这里我们显式写出使用合理数据初始化数组 B、C 和 D 的循环（没有必要初始化 A 这是因为在写之前它不被读）。如果没有为 ccNUMA 系统优化的 OpenMP 应用代码原型运行在几个不同局部区域上，那么当工作集不能刚好放入 cache 中时，将不能获取单个 LD 上的最大性能。这是因为初始化循环被单一线程执行，首次写入 B、C 和 D。因此，所有属于数组的内存页面都被映射到单一的 LD 上。如图 8-4 显示，后果很明显：如果工作集大小刚好适合 cache，则扩展性很好。但是，对于一些大数组来说，8 个线程性能（实圆圈表示）甚至低于 2 个线程（一个 LD）的值（虚方块表示），这是因为所有线程都通过 HT 网络访问 LD0 中的内存，从而导致严重冲突的问题。如上所述，可以通过并行初始化数组来避免这个问题。代码的第 4 ～ 6 行循环可以被如下代码替代：

```
1 ! initialization
2 !$OMP PARALLEL DO
3   do i=1,N
4     B(i) = i; C(i) = mod(i,5); D(i) = mod(i,10)
5   enddo
6 !$OMP END PARALLEL DO
```

这个简单修改，对于UMA系统来说无用，但是对于内存约束情形的ccNUMA系统来说能有很大提升（见图8-4中的圆圈和插图）。当然，当N很大而不能使工作集放入单个局
187 部区域时，数据将以不可控制的方式“自动”地分布式存放。这种效果绝不依赖于当数据分布是关键的情形。

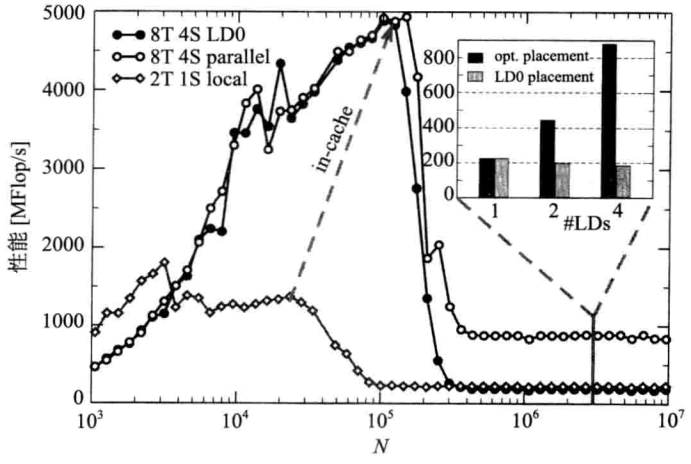


图8-4 在图8-3中（HP DL585 G5）含有4个LD的ccNUMA机器向量性能和扩展性，这里比较8个线程，页面放置在LD0中（实圆圈）和首次访问的正确并行（虚圆圈）。单个LD局部访问的性能数据在此展示以便引用（虚方块）。每个槽的两个线程贯穿始终。cache内扩展性并不受到不合适页布局的影响。对于内存约束情况，将所有数据放在单个LD中有严重后果（请看插图）

有时候没有用来并行的循环，初始化数组并行并不足够。在图8-5左边的OpenMP代码中，第8行A的初始化是通过READ语句在串行区域实现的。右边代码通过并行初始化A纠正了这个问题，和随后访问的方式相同来首次访问它们的数据。虽然READ操作依旧串行，但是数据将分布在不同的局部区域。数组B不需要初始化，但它能够自动正确映射。

要想首次访问方式能够正确工作并且获得很好的扩展循环性能，则必须满足一些条件。

```
1  integer,parameter:: N=1000000          integer,parameter:: N=1000000
2  double precision :: A(N),B(N)          double precision :: A(N),B(N)
3                                         !$OMP PARALLEL DO
4                                           do i=1,N
5                                             A(i) = 0.d0
6                                           enddo
7  ! executed on single LD                !$OMP END PARALLEL DO
8  READ(1000) A                  ------>  ! A is mapped now
9  ! contention problem                   READ(1000) A
10 !$OMP PARALLEL DO                      !$OMP PARALLEL DO
11   do i = 1, N                            do i = 1, N
12     B(i) = func(A(i))                      B(i) = func(A(i))
13   enddo                                  enddo
14 !$OMP END PARALLEL DO                  !$OMP END PARALLEL DO
```

图8-5 通过正确NUMA布局来优化。左边：READ语句仅被单个线程执行，将A放在单个局部区域里。右边：并行初始化得到正确的不同局部区域的页面分布

❑ 初始化的OpenMP循环调度和工作循环必须相同和可重复，也就是，只有一种可能

选择是具有固定大小块组的 STATIC，并且排除任务的使用。因为 OpenMP 标准没有定义默认调度，所以在所有并行循环上显式确定是不错的做法。当前所有的编译器都选择 STATIC 作为默认调度。当然，使用静态调度会给消除负载不均的优化带来一些限制。唯一的简单选择是选择合适大小的块组（尽可能小，但是至少有几个页的大小）。更多关于 ccNUMA 条件下的动态调度的详细信息请参考 8.3.1 节。 188

- 对于有相同迭代次数和相同并行线程的连续并行循环，每个线程应该获取迭代空间同样的部分。OpenMP 3.0 中保证这种行为当且仅当循环使用相同块组大小的 STATIC 调度（或者没有）并且绑定在同一并行区域里。虽然后一种条件往往不能满足，至少无法对程序中所有循环做到，但是现在的编译器生成的代码能够保证循环迭代空间有相同长度并且 OpenMP 调度总是被相同方式分解，甚至在不同循环区域中。
- 硬件必须能够支持将内存带宽扩展到不同局部区域。当然并不总是这样，例如，如果 cache 一致性阻塞产生 NUMA 网络冲突。

遗憾的是并不总是程序员考虑怎样和什么时候首次访问数据。在 C/C++ 中，全局数据（包括对象）在 main() 函数之前初始化。如果不能避免全局，适当对全局数据的局部映射是可能的解决方案，其中代码特征是以允许的通信和计算比衡量 [O68]。关于 OpenMP 和 C++ 组合导致的问题讨论可以参考 8.4 节、[C100] 和 [C101]。

关于分配内存和初始化后的页面如何失去它的页表项没有很好的说明。多数情况下，对处在堆上的内存能充分释放（Fortran 使用 DEALLOCATE，C 使用 free()，C++ 使用 delete[]）。这是为什么上面描述的向量基准采用动态内存的原因。如果随后新的内存块被分配，那么首次访问策略正常应用。尽管如此，一些运行时库优化实现实际上并没有使用 free() 释放内存，而是添加一些页面到“内存池”，从而最后用很小的开销释放。为了避免疑惑，系统文档应该采取相关方法改变这种行为。 189

在共享内存并行代码中局部性问题一般是最突出的。如果独立运行的进程和它们初始化数据时所在的局部区域保持关联，那么它们将自动采取首次访问内存布局策略。请参考 8.3.2 节中可能阻碍局部访问的影响。

8.1.2 通过其他方式的局部性访问

除了基本的首次访问初始化策略，操作系统经常会提供显式页面布局和诊断的高级工具。这些工具自然很不易用。通常是命令行工具或者通过可配置动态对象来影响内存分配和首次访问行为而不需要改变源代码。典型功能包括：

- 设置策略或者首先项来将内存页映射到特定的局部区域而不用考虑所分配的进程或者线程运行在什么地方。
- 为跨局部区域连续访问页面的映射设置“循环”或者随机策略。如果共享内存并行程序没有固定的访问模式（例如，受到负载均衡的限制），并且无法采用一致性首次访问映射，这将使得内存约束代码的扩展性并行处于低级别。参见 8.3.1 节的相关实例。
- 可能以单个进程为基础，诊断页面在局部性区域上的当前分布。

除了单独工具，还存在提供更好的细粒度控制页面布局的文档化 API 的库。在 Linux OS 中，numatools 工具包包含所有上面描述的功能，并且还允许线程或者进程关联工具（也就是，决定线程或者进程应该运行在哪）。详细信息请参考附录 A。

8.2 案例分析：稀疏 MVM 的 ccNUMA 优化

在初始化的时候代码工作集内存页面被映射到单个局部区域将导致冲突问题，所以很显
190 然在 ccNUMA 系统上 OpenMP 并行化的稀疏 MVM 具有很差的扩展性（请参考图 7-7）。通过编写采取首次访问映射策略的并行初始化循环，扩展性得到很大提高。既然对于 JDS 策略一样，所以这里我们仅探讨 CRS。数组 C、val、col_idx、row_ptr 和 B 必须并行初始化：

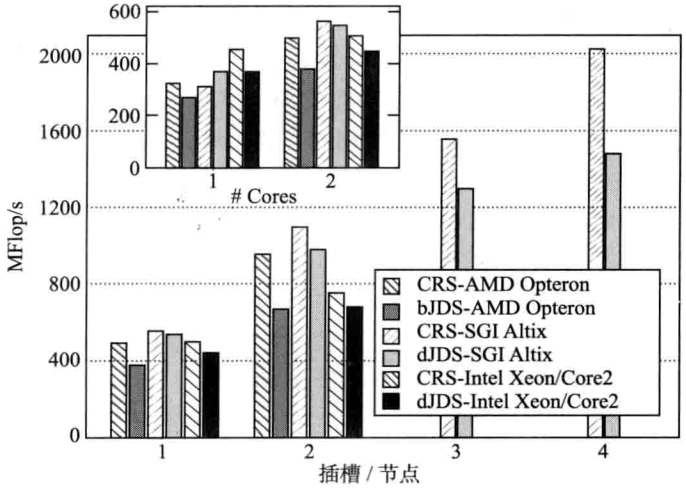


图 8-6 比较 CRS（阴影条状）和块化 JDS（实条状）在三个不同架构上的 ccNUMA 优化的 OpenMP 并行化稀疏 MVM 性能和强扩展性。图 7-7 是没有合理布局的性能。不同扩展基已经被分开（主框图是单槽或者单个 LD，插图是单核）

```
1  !$OMP PARALLEL DO
2    do i=1,Nr
3      row_ptr(i) = 0 ; C(i) = 0.d0 ; B(i) = 0.d0
4    enddo
5  !$OMP END PARALLEL DO
6  .... ! preset row_ptr array
7  !$OMP PARALLEL DO PRIVATE(start,end,j)
8    do i=1,Nr
9      start = row_ptr(i) ; end = row_ptr(i+1)
10     do j=start,end-1
11       val(j) = 0.d0 ; col_idx(j) = 0
12     enddo
13   enddo
14 !$OMP END PARALLEL DO
```

B 的初始化是基于矩阵的非零值基本上聚集在主对角线附近的假设。由于依赖于矩阵结
191 构，所以在实践中对 RHS 向量来说很难采取合适的布局策略。

图 8-6 性能数据和图 7-7 采用了同一架构和 MVM 代码，只不过是加入了合适的 ccNUMA 布局。正如所期待的，对于 UMA 平台来说，可扩展性是没什么变化的，对于 ccNUMA 系统上的双线程来说也如此（请参考插图）。原因是两个架构都有 UMA 类型的两个处理器局部区域。对于四个线程以及以上情况，局部性优化将产生巨大的性能提升。特别对于 CRS，当线程从 $2n$ 增加到 $2(n+1)$，它的扩展性近于完美值（主框图中的扩展基线分别是局部区域或者槽）。JDS 也能从优化中获取性能提升，但是当线程数很大时落后于 CRS。

这是因为 JDS 的置换映射使其很难放置更大部分的 RHS 向量到正确的局部区域，因此导致了 NUMA 通信量增加。

8.3 页面布局缺陷

我们已经证明对于 ccNUMA 架构包括广泛使用的双槽集群节点来说数据布局是第一重要的。原则上，ccNUMA 对内存约束的代码提供了优秀的扩展性，但是 UMA 系统更容易控制并且不需要为局部性访问提供额外的代码优化。正如所期望的一样，ccNUMA 设计在性价比很高的双槽配置 HPC 市场盛行。但是需要强调的是 8.1 节介绍的优化布局并不总是适用的，例如，当动态调度不可避免时（请参考 8.3.1 节）。此外，可能会得到内存布局问题仅限制于共享内存编程的结论；而实际并不是如此，进一步理解请参考 8.3.2 节。

8.3.1 非 NUMA 友好的 OpenMP 调度

正如 6.1.3 节和 6.1.7 节介绍，动态或者指导性调度和 OpenMP task 结构在一些负载均衡很差的情况下，如果频繁为线程分配任务的额外开销可以忽略的话，相对于静态工作分布来说更好。另一方面，如果线程组在不同局部区域上时，任何形式的动态调度（包括分任务）将导致扩展问题。对线程的任务分配是不可预测的，甚至每次运行都不同，这将排除“优化”页面布局策略。

在这样情形下同时丢掉并行首次访问策略没有解决方案，这是因为性能将受制于单个内存接口。为了获取所获得最大带宽的大部分，最好在局部区域之间以循环方式部署工作集的内存页并且希望静态访问分布。向量三元组仍然作为了解随机页面访问作用的方便工具。我 192
们通过强制使用具有页面大小的块组的静态调度来修改初始化循环（假设 4KB 的页面）。

```
! initialization
!$OMP PARALLEL DO SCHEDULE(STATIC,512)
  do i=1,N
    A(i) = 0; B(i) = i; C(i) = mod(i,5); D(i) = mod(i,10)
  enddo
!$OMP END PARALLEL DO
  ...
  do j=1,R
!$OMP PARALLEL DO SCHEDULE(RUNTIME)
    do i=1,N
      A(i) = B(i) + C(i) * D(i)
    enddo
!$OMP END PARALLEL DO
    call dummy(A,B,C,D)
  enddo
```

通过设置 OMP_SCHEDULE 环境变量可以测试不同循环调度策略。图 8-7 显示当块组大小为 c 时，静态和动态调度的并行性能对比，使用的是来自图 8-3 的 8 线程 4 槽的 ccNUMA 系统。当 c 很大时，也就是单个块组占据几个内存页，性能近似于随机访问所有 LD 的情况，而与采取什么调度策略无关。在这种情形下，一个线程所需要的 75% 页面位于远程区域。尽管这种不定模式能够获得一定程度的并行（相对于虚线显示的纯串行初始化情况），但是相对于理想情况（实线）仍然有几乎 50% 的性能损失。当 $c = 512$ 时需要注意：使用静态调度，三元组循环访问模式和初始化循环的内存布局策略相匹配，使得（绝大多数）局部性访问每个 LD。与最佳可能结果的剩余差别应该归因于无法和页边界对齐的数组，这 193
将导致一些不确定性。注意操作系统和编译器经常为可配置边界（候选者有 SIMD 数据类型

长度、高速缓冲行和内存页面等）提供对齐数据结构的手段。但是需要注意以避免强对齐数据结构的走样。

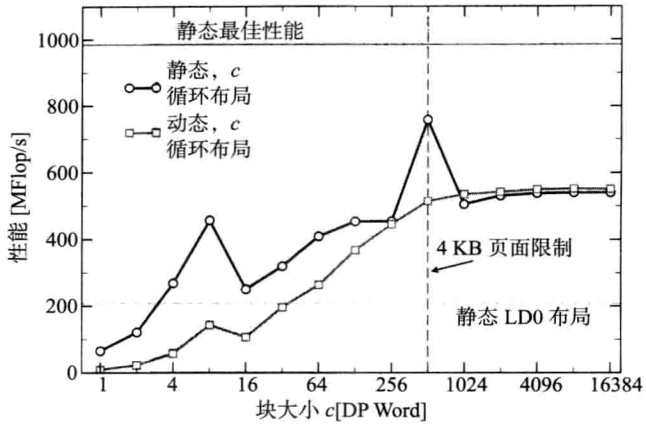


图 8-7 8 向量三元组性能对比。在四个 LD 的 ccNUMA 系统上（请参考图 8-3）8 线程的静态调度和动态调度。页面布局特意采取的是循环方式。显示最佳并行布局和静态调度的 LD0 布局的性能以便引用

尽管与 NUMA 作用不直接相关，但分析小规模块组的情形还是有指导意义的。和静态调度相比动态调度所引起的额外开销导致很大差距。如果 c 小于 cache 行长度（64 字节），cache 失效时，虽然仅需要一小部分，但是还需要传输整个 cache 行，因此这是 $c \leqslant 64$ 的特有行为。当 c=16 时下降的解释以及性能逐渐增长直到页面大小时的解释留作练习（请看本章结尾习题）。

总之，如果排除纯静态调度（不含块大小），循环布局策略至少可以利用一些并行性。如果可能，含有合适大小块组的静态调度应该作为 OpenMP 工作共享循环调度方案以避免调度带来的过多开销。

8.3.2 文件系统 cache

尽管遵循所有关于关联和页面布局注意事项，不但 OpenMP 程序仍然存在可扩展可能，而且独立运行进程的整个系统性能还是低于预期。硬盘 I/O 操作会使得操作系统建立存储最近读或写的文件数据的高速缓冲区以便复用。高速缓冲区的大小和布局很大程度上与系统相关，并且可以配置，但是默认设置适用于大多数情况，虽然可以利于实行好的 I/O 性能，但是对于 ccNUMA 局部性来说不利。

请参看图 8-8 的例子：在 LD0 上运行的一个线程或者进程写一个大文件到硬盘，并且操作系统为文件系统高速缓冲区预留

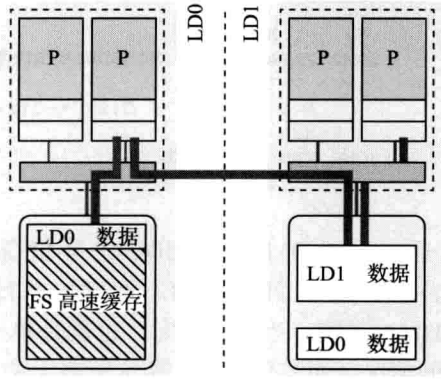


图 8-8 文件系统 cache 能够阻止局部性访问页面被放置在局部区域，这样将导致非局部访问以及冲突问题。这里显示的是局部区域 0，在这个区域上 FS 高速缓存使用局部内存的绝大数空间。被核所分配和初始化的页面在 LD0 的页面映射到 LD1

了一些这个区域的内存空间。随后，在这个区域的相同或者另一个进程分配和初始化内存（“LD0 数据”），但是不是所有页面都刚好放入已经有高速缓冲区的 LD0 空间。默认情况下，许多系统将多余页面映射到另一个局部区域，以便从程序员角度来说首次访问策略是正确的，然而 LD0 数据的非局部访问以及 LD1 内存接口冲突仍然会出现。 194

一个简单试验证明了这种效果。我们比较一个 UMA 系统（图 4-4 中双核双槽的 Intel Xeon 5160）和一个两个 LD 的 ccNUMA 节点（图 4-5 中双核双槽 AMD Opteron），两者都是 4G 内存。在两个系统上的一个循环里我们执行如下步骤：

1）向硬盘写入“脏”数据并使得所有高速缓冲区不合法。这一步很大程度依赖系统；通常存在一个管理员执行的程序来实现⊖，或者供应商提供的库为来提供选择。

2）向局部硬盘写入大小为 S 的文件。文件最大值等于内存大小。S 应该从很小的值开始，并且在循环的每次迭代后增加直到等于系统的内存。

3）同步高速缓冲区以便在后台没有刷新操作（但是高速缓冲区还是填满的）。这通常使用标准 UNIX 的 sync 命令完成。

4）使用合适的关联机制（请参考附录 A）在每个核上运行相同的向量三元组基准。所有工作集的大小应该等于节点内存一半大小。整体性能是以 MFlop/s 和 S 大小（这里等于高速缓冲区大小）的比值为单位。

结果显示在图 8-9 中。在 UMA 节点上，因为没有局部性概念，所以高速缓冲区没什么影响。另一方面，ccNUMA 系统显示了随着大小增加而出现的很强的性能下降，并且当 LD 被高速缓冲区填满时（大约 2GB），性能降到最低：发生于所有在 LD0 中被三元组循环初始化的内存页面被映射到 LD1。如果文件变得更大，则性能又上升，这是因为单个局部区域太小甚至连高速缓冲区都放不下。如果文件大小等于内存大小（4GB），那么并行首次访问策略按照需要更新高速缓冲区页面并且像正常方式一样工作。 195

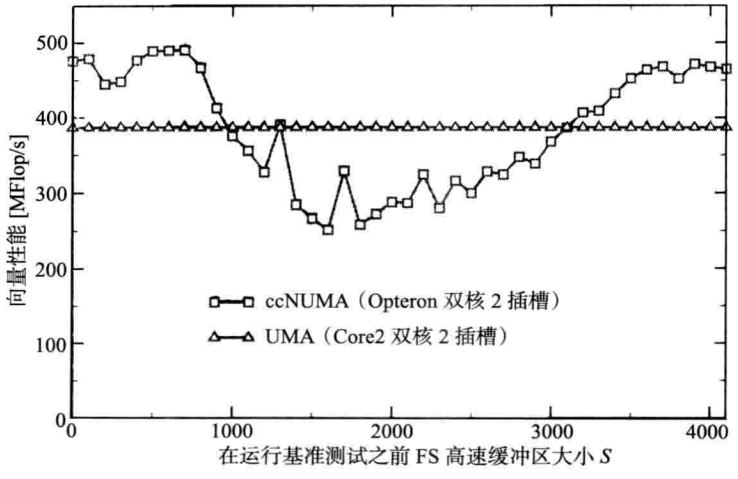


图 8-9 当运行四个并发向量三元组程序时，在 ccNUMA 和 UMA 系统上大文件缓冲区对性能影响对比（都有两个槽和四个核以及 4GB 的 RAM）。缓冲区被单核填满。细节请参考文章 (Michael Meier 的基准数据)

⊖ 在当前 Linux OS 上，这一步骤通过执行 echo1>/proc/sys/vm/drop_caches 命令来完成。SGI Altix 系统提供了 bcfree 命令行，该命令行提供了类似的功能。

从这个实验中我们可以学到许多经验。最重要的是它证明了在 ccNUMA 上的局部性问题既不受 OpenMP（或者一般的共享内存并行）程序限制，也不是通过修正的首次访问策略来获得“最佳”扩展性。高速缓冲区甚至可能是上次其他用户运行工作留下来的。理想情况下，当一项工作结束时，HPC 中应该存在一种方式能够自动的“更新”高速缓冲区以便为下一个用户留下“干净”机器。万不得已的情况下，如果这里没有用户级的工具，并且系统管理员没有对这个问题给予应有的重视，那么没有权限的普通用户可以执行分配和初始化所有内存的“扫描”代码。

```
1   double precision, allocatable, dimension(:) :: A
2   double precision :: tmp
3   integer(kind=8) :: i
4   integer(kind=8), parameter :: SIZE = SIZE_OF_MEMORY_IN_DOUBLES
5   allocate A(SIZE)
6   tmp=0.d0
7  ! touch all pages
8  !$OMP PARALLEL DO
9    do i=1, SIZE
10     A(i) = SQRT(DBLE(i))   ! dummy values
11   enddo
12 !$OMP END PARALLEL DO
13 ! actually use the result
14 !$OMP PARALLEL DO
15   do i=1,SIZE
16     tmp = tmp + A(i)*A(1)
17   enddo
18 !$OMP END PARALLEL DO
19   print *,tmp
```

这个代码可以作为用户应用的一部分来更新被运行程序 I/O 填充的高速缓冲区（类似读写）。第二个循环的唯一目的是阻止编译器优化，这是因为它发现 A 从来没有被使用。并行化循环当然是可选的但是可以加速整个进程。注意，由于依赖于内存实际大小和“脏”文件高速缓冲块大小，这个过程可能花费大量的时间：最坏的情况下，几乎所有的主存不得不写回硬盘。

高速缓冲区和相应的局部性问题是在 ccNUMA 节点的集群上运行并行程序的性能倾向于波动的一个原因。如果很多节点参与，则仅有一个节点的大高速缓冲区可能阻塞整个并行
196 应用程序的性能。采取给定环境的所有可能的选项来构建环境以便减少高速缓冲区的影响是系统任务管理员的任务。例如，一些系统允许配置哪些缓冲页面保持，赋予局部内存高优先级和按照需要更新缓冲区。

8.4 C++ 中的 ccNUMA 问题

如上所示，只要基本的内存访问模式被识别，那么内存访问的局部性经常采用像 Fortran 或者 C 这样的语言来实现。然而由于 C++ 面向对象的特点，它完全又是另一回事[C100,C101]。在这个小节中，我们想指出在 ccNUMA 系统上使用 OpenMP 并行 C++ 代码的常见缺陷。

8.4.1 对象数组

当使用 new[] 操作符为类型为 D 的对象分配数组时，最基本的问题会出现。为了简单起见，我们选择 D 作为含有必要重载操作符的 double 包装类使它更像一个基本类型：

```
class D {
  double d;
public:
  D(double _d=0.0) throw() : d(_d) {}
  ~D() throw() {}
  inline D& operator=(double _d) throw() {d=_d; return *this;}
  friend D operator+(const D&, const D&) throw();
  friend D operator*(const D&, const D&) throw();
  ...
};
```

假设所有操作符都被正确实现，D 和 double 的唯一区别是类型 D 的对象实例化导致立即初始化，而 double 不同，也就是，在 a=new D[N] 中，内存分配如常，但是每个数组成员的默认构造函数被调用。因为 new 对于 NUMA 没有任何了解，所以这些调用是被执行 new 的线程完成。因此所有数据均终止于线程的局部内存中。这个方式的有一个问题是默认构造函数并不访问数据，但是这并不是必须的或者需要的。

程序员应该首先将数组数据所使用的内存页面映射到正确的节点上以便每个线程能够局部访问，然后调用构造函数初始化对象。这将使用 new 布局来完成，这时创建的对象数量以及实例化的精确地址（基址）被确定。new 布局并不调用构造函数。使用布局 new 的一个简单方式就是重载 D::operator new[]。这个操作符单独负责分配“原始”内存。一个 197
重载版本除了内存分配，还能够为好的 NUMA 布局平行初始化页面（我们忽略失败时抛出 std::bad_alloc 的需要）：

```
void* D::operator new[](size_t n) {
  char *p = new char[n];        // allocate
  size_t i,j;
#pragma omp parallel for private(j) schedule(runtime)
  for(i=0; i<n; i += sizeof(D))
    for(j=0; j<sizeof(D); ++j)
      p[i+j] = 0;
  return p;
}

void D::operator delete[](void* p) throw() {
 delete [] static_cast<char*>p;
}
```

使用布局 new，C++ 运行时能够自动在正确的位置构建数组中的所有对象。注意，C++ 运行时通常需要比所有对象大小累加和多一些空间来为实际数据存放管理信息。因为这部分数据相对于 NUMA 相关数组来说很小，所以没有什么显著影响。

像上面 D 类一样简单地重载操作符 new[]。但是动态成员是有问题的，这是因为它们的 NUMA 局部性不能够很好地受控制：

```
class E {
  size_t s;
  std::vector<double> *v;
public:
  E(size_t _s=100) : s(_s), v(new std::vector<double>(s)) {}
  ~E() { delete [] v; }
  ...
};
```

E 的构造函数初始化 E::s 和 E::v，并且它们将是在 E 数组构造时，通过重载 E::operator new[] 来服从于 NUMA 布局的仅有数据。E::v 的内存地址不被这种机制控制；事实上，在 STL 内部使用 double() 对象的拷贝时，std::vector<double> 被预先设置。这是在 E::operator

new[] 执行后发生在 C++ 运行时里的。所有内存被映射到一个局部区域。

如果坚持使用 new[] 构造数组对象，那么当使用标准 C++ 和 STL 结构时，这种情形几乎不可能避免。最好的建议是在循环中显式调用对象构造函数，并且使用一个仅存储指针的容器：

```
1   std::vector<E*> v_E(n);
2
3 #pragma omp parallel for schedule(runtime)
4   for(size_t i=0; i<v_E.size(); ++i) {
5     v_E[i] = new E(100);
6   }
```

198

既然类构造函数由不同线程并发调度，那么肯定是线程安全的。

8.4.2 标准模板库

前面展示的 C 风格数组的操作对于 C++ 来说是不行的；STL 的 std::vector<> 容器更安全并且更方便，但是关于 ccNUMA 页面布局也有自己的问题。即使对于像 double 这样有默认构造函数的简单数据类型，布局问题仍然存在，例如，std::vector<>(int) 中对象所分配的内存是使用 std::uninitialized_fill() 并用 value_type() 副本来填充。专门设计一个能够适应 NUMA 的容器类会允许更多的优化手段，但是 STL 定义了一个 allcators 自定义抽象层，它能够高效地对容器内存管理底层的细节进行封装。通过使用它们，很多使用 std::vector<> 的程序只需少量修改就能保证正确 NUMA 布局。

STL 容器有一个可以设置使用何种分配器类的可选模板参数 [C102, C103]。默认情形是 std::allocator<T>。一个分配器类将提供这些方法（类的命名空间略去）：

```
1 pointer allocate(size_type, const void *=0);
2 void deallocate(pointer, size_type);
```

这里 size_type 是 size_t，pointer 是 T*。allocate() 方法被容器构造函数调用来像 operator new[] 一样为数组对象构建内存。然而，因为所有相关补充信息都存放在额外的成员变量中，所以分配的内存仅能够匹配容器内容所需空间，至少在初始化构建中（请参考下面）。allocate() 的第二个参数能够为分配器提供额外信息，但是它的语义不是标准的。deallocate() 负责释放内存。

最简单的适应 NUMA 的分配器应该能够做到 allocate() 不仅能够分配内存而且能够并行的初始化。作为参考，代码清单 8-1 显示了一个简单的、NUMA 友好的分配器代码，这里使用标准 malloc() 来分配内存。在第 19 行，OpenMP API 函数 omp_in_parallel() 用来决定分配器是否被并行区域调用。如果是，初始化循环将被跳过。使用这个模板，std::vector<> 对象被建立时，第二个模板参数都要被确定：

199

```
1   vector<double, NUMA_Allocator<double> > v(length);
```

代码清单 8-1 一个 NUMA 分配器模板。实现在某种程度上依据 C++ 标准需求进行了简化

```
1 template <class T> class NUMA_Allocator {
2 public:
3   typedef T* pointer;
4   typedef const T* const_pointer;
5   typedef T& reference;
6   typedef const T& const_reference;
7   typedef size_t size_type;
8   typedef T value_type;
9
```

```
10    NUMA_Allocator() { }
11    NUMA_Allocator(const NUMA_Allocator& _r) { }
12    ~NUMA_Allocator() { }
13
14    // allocate raw memory including page placement
15    pointer allocate(size_type numObjects,
16                     const void *localityHint=0) {
17      size_type len = numObjects * sizeof(value_type);
18      char *p = static_cast<char*>(std::malloc(len));
19      if(!omp_in_parallel()) {
20  #pragma omp parallel for schedule(runtime) private(ofs)
21        for(size_type i=0; i<len; i+=sizeof(value_type)) {
22          for(size_type j=0; j<sizeof(value_type); ++j) {
23            p[i+j]=0;
24          }
25        }
26      return static_cast<pointer>(m);
27    }
28
29    // free raw memory
30    void deallocate(pointer ptrToMemory, size_type numObjects) {
31      std::free(ptrToMemory);
32    }
33
34    // construct object at given address
35    void construct(pointer p, const value_type& x) {
36      new(p) value_type(x);
37    }
38
39    // destroy object at given address
40    void destroy(pointer p) {
41      p-> value_type();
42    }
43
44  private:
45    void operator=(const NUMA_Allocator&) {}
46  };
```

200

在内存分配之后所做之事和数组对象很相似，也有同样的限制：分配器的 construct() 方法被每个对象调用，并且使用 new 布局策略在正确的地址构建每个对象（36 行）。在销毁对象时，每个对象的析构函数通过第 41 行的 destroy() 方法显式调用（极少的情形下是必需的）。注意容器构造和析构不仅是 construct() 和 destroy() 函数调用的地方，并且还有很多立即消除 NUMA 局部性的方法。例如，根据容器的大小相比于容量的概念，在一个“满”容器上调用 std::vector<>::push_back() 将导致重新分配所有内存，同时增加空间，然后将原来的对象复制到新的位置。NUMA 分配器将执行首次访问布局策略，对于使用容器新的容量而不是对于容器大小来说如此。因此，布局几乎是次优的。程序员需要牢记并不是 std::vector<> 的所有功能都适合使用 ccNUMA 平台。我们这里将不介绍其他 STL 容器（deque、list、map、set 等）。

顺便提一句，具有相同类型、符合标准分配器的对象必须总是会比较是否相等 [C102]：

```
1  template <class T>
2  inline bool operator==(const NUMA_Allocator<T>&,
3                         const NUMA_Allocator<T>&) { return true; }
4  template <class T>
5  inline bool operator!=(const NUMA_Allocator<T>&,
6                         const NUMA_Allocator<T>&) { return false; }
```

这将导致分配器对象必然是无状态的，同时排除了一些可以想到的优化策略。必须提供

T=void 的模板特化（这里没有显示）。在文献总会探讨这种情况以及其他的特性。不使用简单 malloc() 而使用更复杂方法的策略当然存在。

总之，这里展示的方法有助于更改已有的 C++ 程序，使其适应 ccNUMA 而没有太多困难。当然新设计的代码应该开始就使用 ccNUMA 的并行化。

习题

8.1 *动态调度和 ccNUMA*。当一个内存约束、OpenMP 并行代码运行在 ccNUMA 系统的所有槽上时，应当使用静态调度和并行的初始化数据以保证内存访问绝大多数都是局部的。我们想分析如果静
201 态调度不作为选择，例如为了负载均衡的原因，将会有什么样情形。

对于有两个局部区域的系统，计算内存约束并行循环的动态调度时期望的性能影响。为了简单起见，假设每个 LD 上只运行一个线程（核）。这个线程能够以性能 p 占满局部或者远程内存总线。LD 间的网络应该无限快，也就是，在 LD 间的链接上非局部传输没有开销并且没有冲突影响。进一步假设所有页面都是同样地分布在整个系统上并且动态调度是随机的（也就是，每个线程以一种等概率的随机方式访问所有 LD）。最后，假设块组足够大以便对硬件预取或者 cache 行部分使用没有坏的影响。

静态调度和最佳负载均衡的代码性能是 $2p$。在动态调度下期望的性能是多少（也是最佳负载均衡）？

8.2 *令人遗憾的块组大小*。当块组大小在 16 ～ 256 之间时，是什么原因造成并行向量三元组在四个 LD 的 ccNUMA 机器（见图 8-7）上性能下降？提示：内存页作用很关键。

8.3 *加速“小”作业*。如果一个 ccNUMA 系统利用率很低，例如，如果这里线程数量少于局部区域，并且它们都执行（内存约束）代码，那么首次访问策略仍然是最好的页面布局策略吗？

8.4 *三角矩阵 – 向量乘法*。使用 OpenMP 并行化一个三角矩阵向量乘法：

```
1 do r=1,N
2   do c=1,r
3     y(r) = y(r) + a(c,r) * x(c)
4   enddo
5 enddo
```

这里核心的并行性能问题是什么？一般如何解决，并且对于 ccNUMA 系统有什么需要特殊注意的？你可以忽略标准的扩展优化（循环展开、块化）。

8.5 *重载 NUMA 页面布局*。在 8.4.1 节中，我们通过重载 D::operator new[] 为 D 类型的数组对象强制使用 NUMA 布局。在适应 NUMA 的分配器类中也有类似的事情（见代码清单 8-1）。为什么我们
202 使用嵌套循环初始化而非 i 上的单一循环？

使用 MPI 进行分布式存储并行内存编程

自从并行计算机进入高性能计算市场以来，关于什么编程模型才是最适合并行机的讨论一直十分激烈。显式地使用消息传递（Message Passing，MP），即进程间通信，无疑是既乏味又复杂而同时也最灵活的并行化方法。并行机生产商意识到了市场对于高效消息传递设备的需求，因此随机器提供了专门的不可移植库，这种情况一直延续到 20 世纪 90 年代。在那时，需要制定一个联合标准使得科研人员能够很轻松地编写出可以跨平台的并行程序成为共识，MPI（Message Passing Interface）——消息传递接口因此应运而生。到目前为止，MPI 标准已经有过几次扩展，一些免费和商用的 MPI 实现 [W125,W126,W127] 也均支持它。除了包含通信程序外，MPI 也包含有效的平行 I/O 工具（前提是底层硬件支持）。现在，在任何高性能计算机系统的安装中，MPI 库都被当作是一个不可缺少的成分，众多类型的互联均被它所支持。

目前的 MPI 标准 MPI 2.2（我们在本书中一直提到的版本）定义了 500 多个函数，完全介绍它们显然已经超出了本书的范围。本章我们重点介绍消息传递的重要概念和几个特定的 MPI 函数，同时介绍一些知识方便读者查阅一些最新的参考资料 [P13,P14] 以及 MPI 标准文档 [W128,P15]。

9.1 消息传递

对于分布式存储类型的并行机来说，一台处理器无法直接访问另一台处理器的地址空间，因此需要借助消息传递。消息传递也可以被看成是一个编程模型（programming model）并使用在共享内存或者混合型的计算机系统上（参见第 4 章）。作为当今主流的消息传递标准，MPI 遵循以下规则：

- 同一程序运行在全部进程上（单程序、多数据，也称 SPMD）。相比于更一般的 MPMD（多程序、多数据）模型，这里并没有约束，因为参与并行计算的所有进程可以用称为 rank 的唯一标识符来区分（参见下面）。如今的大多数 MPI 实现版本均支持以不同的二进制文件发起的不同进程。MPMD 型的消息传递库称为并行虚拟机（Parallel Virtual Machine, PVM）[P16]。由于在近些年它的重要性已经逐渐下降，这里将不再提及。
- 程序用串行语言编写，如 Fortran、C、C++。数据交换，即消息的发送和接受，通过调用一个恰当的库来完成。
- 一个进程中的所有变量对这个进程来说都是局部的，这里没有共享内存的概念。

需要补充的是对于分布式存储计算机，消息传递并不是唯一的编程范式。专门化的语言如 High Performance Fortran（HPF）、Co-Array Fortran（CAF）[P17]、Unified Parallel C（UPC）[P18] 等已经被开发出来。它们内置了分布式存储的并行化，但是它们并没有发展成广阔的用户社区，而且它们能否达到 MPI 的效率也不清楚。

在一个消息传递程序中，消息在进程间携带数据。这些进程可以各自运行在独立的计算节点上，抑或同一个节点上的不同核，甚至可以分时运行在同一个处理器核上。消息可以是一个简单的条目（如一个双精度字），也可以是一个分散到整个地址空间的复杂结构。对于一个使用有序方式传递的消息，需要提前确定如下参数：

- 哪个进程在发送消息？
- 数据在发送进程上的什么位置？
- 发送的是哪种数据？
- 有多少数据？
- 哪个进程将要接收消息？
- 数据将被放在接收进程的什么位置？
- 接收进程准备接收的数据量是多少？

所有实际传输数据的 MPI 调用都必须以某种方式指定这些参数。注意以上这些参数都严格地与点对点的通信有关，即恰好有一个发送者和一个接收者。正如我们即将看到的，MPI 不仅支持在两个进程间发送单一的消息，同时对于那些更复杂的情况也含有一个类似于上面参数的集合。

MPI 是一个十分广泛的标准，它包含数量庞大的库例程。不过幸运的是，大部分应用只
204 需要它们中很少的一部分。

代码清单 9-1 Fortran 90 编写的一个简单且功能完善的“Hello World”MPI 程序

```
1  program mpitest
2
3  use MPI
4
5  integer :: rank, size, ierror
6
7  call MPI_Init(ierror)
8  call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
9  call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)
10
11 write(*,*) 'Hello World, I am ',rank,' of ',size
12
13 call MPI_Finalize(ierror)
14 end
```

9.2 MPI 简介

9.2.1 一个简单例子

MPI 作为一个库很容易获得。在编译和链接 MPI 程序时，编译器和链接器需要编译选项来指定包含文件（即 C 头文件和 Fortran 模块）和库文件可以在何处找到。由于安装的地址可能会有很大变化，大多数 MPI 实现提供了包装好的编译器脚本（通常称为 mpicc、mpif77 等）。它们自动支持所需的编译选项，除此之外，它们的行为与正常的编译器十分类似。MPI 标准并没有规定 MPI 程序该如何编译和执行，所以请务必查阅相关系统文档。

代码清单 9-1 展示了一个用 Fortran 90 编写的“Hello World”MPI 程序。（C 语言版本请参见代码清单 9-2，我们将主要介绍 Fortran 版的 MPI 绑定，在恰当的地方仅描述和 C 语言相比的不同之处。尽管标准中也定义了 C++ 版本的绑定，但可用性有限，因此这里将不再涉及。事实上，在 MPI 2.2 版本中，C++ 已经被弃用了。）代码第 3 行，导入 MPI 模块，

它提供了所需的全局变量和定义（在 C 语言中，预处理程序用来读取 mpi.h 头文件，Fortran 77 中等价的头文件称为 mpif.h）。所有的 Fortran 版 MPI 调用都使用一个 INTENT(OUT) 参数，这里称为 ierror。它传递对用户代码使用 MPI 操作是否成功的信息，值 MPI_SUCCESS 意味着没有错误。在 C 语言中，函数的返回值用来实现这一功能，故不存在 ierror 参数。由于失败跳出并没有包含在当下的 MPI 标准中，同时检查点或重新启动等功能在用户代码中通常都有实现，因此在实际操作中，检错代码几乎很少用到。

除了变量声明外，任何 MPI 程序代码的第一句都是调用 MPI_Init()，用以初始化并行环境（第 7 行）。若某一种线程的并行同 MPI 一起使用，那么仅调用 MPI_Init() 是不够的，详见第 11 章。 205

代码清单 9-2 C 语言编写的一个简单且功能完善的“Hello World”MPI 程序

```
1  #include <stdio.h>
2  #include <mpi.h>
3
4  int main(int argc, char** argv) {
5    int rank, size;
6
7    MPI_Init(&argc, &argv);
8    MPI_Comm_size(MPI_COMM_WORLD, &size);
9    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10
11   printf("Hello World, I am %d of %d\n", rank, size);
12
13   MPI_Finalize();
14   return 0;
15 }
```

MPI 对 C 语言的绑定遵循区分大小写的命名模式，如 MPI_Xxxxx...，而 Fortran 语言并不区分。相反于 Fortran，C 语言绑定的 MPI_Init() 函数将指针返回到主函数参数，因此 MPI 库可以估计或删除那些有可能由 MPI 发起进程带来的额外命令行参数。

在初始化时，MPI 建立了所谓的世界通信器（*world communicator*），称作 MPI_COMM_WORLD。通信器定义了能被通信句柄引用的一组 MPI 进程。句柄 MPI_COMM_WORLD 描述了所有已经发起的作为并行程序一部分的进程。如果需要，其他通信器可以定义为 MPI_COMM_WORLD 的子集。几乎所有的 MPI 函数调用均需要通信器作为参数。

第 8、9 行的函数调用 MPI_Comm_size() 和 MPI_Comm_rank() 各自决定了并行程序中发起进程的数目（size）以及所调用进程的唯一标识符（rank）。注意，C 语言绑定需要指定输出参数（如上文提到的 rank 和 size）为指针。通信器中的进程排序，如本例中 MPI_COMM_WORLD，是从 0 开始连续递增的。代码第 13 行，调用函数 MPI_Finalize() 以结束并行程序。注意，除进程 0 外，其他 MPI 进程均不可以在调用 MPI_Finalize() 后继续执行任何代码。

为了编译运行代码清单 9-1 中的源代码，通常的实现方式可能需要以下几步：

```
1 $ mpif90 -O3 -o hello.exe hello.F90
2 $ mpirun -np 4 ./hello.exe
```

它将会编译代码然后发起 4 个进程来运行这段程序。在并行程序执行前，这些处理器必须先通过资源管理（批处理）系统来分配任务，MPI 进程具体如何发起完全取决于其实现。理想情况下，启动机制使用资源管理器提供的设施（如运行在所有节点上的守护进程）来发起进程。进程与核之间的关联也是如此。如果此 MPI 实现版本不提供用于 arrinity 控制的直 206

接设施，那么将会使用附录 A 中提到的方法。

这个小程序的输出结果如下：

```
Hello World, I am 3 of 4
Hello World, I am 0 of 4
Hello World, I am 2 of 4
Hello World, I am 1 of 4
```

尽管 MPI 程序的标准输出和标准错误通常会重定向到程序开始的终端，但是如果不使用其他方法强制指定，不同进程输出到达的顺序是不确定的。

9.2.2 消息和点对点通信

实例“Hello World”中除去发起及终止进程外，没有包含任何真正的进程间通信。MPI 消息被定义为一个包含特殊 MPI 数据类型的数组。数据类型可以是基本类型（取决于每种编程语言的标准类型），也可以是由适当 MPI 调用定义的导出类型。MPI 需要知道消息的数据类型的原因在于它支持异构的环境，这种环境在处理动态数据转换时是必需的。对于任何将要进行的消息传递，发送端和接收端的数据类型必须匹配。表 9-1 和表 9-2 分别给出了 C 语言和 Fortran 语言绑定的部分 MPI 数据类型列表。

表 9-1 Fortran 语言绑定的标准 MPI 数据类型

MPI 类型	Fortran 类型
MPI_CHAR	CHARACTER(1)
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_BYTE	N/A

表 9-2 部分 C 语言绑定的标准 MPI 数据类型。适当时存在无符号类型

MPI 类型	C 类型
MPI_CHAR	signed char
MPI_INT	signed int
MPI_LONG	signed long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_BYTE	N/A

当仅有一个发送者和一个接受者时，我们称这种情况为点对点通信。收发两端用各自的进程号来唯一确定。每一个点对点的消息携带一个额外的整数标签，即所谓的 tag，用来确
207 定的消息类型，并且两端必须匹配。它可以携带任何额外信息，当不需要时，它将被设置成常值。从一个进程发送消息到另一进程的基本 MPI 函数为 MPI_Send()：

```
<type> buf(*)
integer :: count, datatype, dest, tag, comm, ierror
call MPI_Send(buf,      ! message buffer
              count,    ! # of items
              datatype, ! MPI data type
              dest,     ! destination rank
              tag,      ! message tag (additional label)
              comm,     ! communicator
              ierror)   ! return value
```

消息缓冲区的数据类型可以改变，MPI 接口及其在模块中声明的协议和头文件均支持

它[⊖]。接收消息时使用函数 MPI_Recv():

```
1  <type> buf(*)
2  integer :: count, datatype, source, tag, comm,
3  integer :: status(MPI_STATUS_SIZE), ierror
4  call MPI_Recv(buf,        ! message buffer
5                count,      ! maximum # of items
6                datatype,   ! MPI data type
7                source,     ! source rank
8                tag,        ! message tag (additional label)
9                comm,       ! communicator
10               status,     ! status object (MPI_Status* in C)
11               ierror)     ! return value
```

与 MPI_Send() 相比，此函数有一个额外的输出参数——status 对象。函数 MPI_Recv() 返回后，status 对象可以用来决定尚没有被调用所固定的参数。它主要适用于消息的长度，因为参数 count 表示的只是接收端可以接收消息的最大长度值，实际接收消息的长度很可能 208
小于 count。调用函数 MPI_Get_count() 可以得到实际的长度值。

```
1  integer :: status(MPI_STATUS_SIZE), datatype, count, ierror
2  call MPI_Get_count(status,     ! status object from MPI_Recv()
3                     datatype,   ! MPI data type received
4                     count,      ! count (output argument)
5                     ierror)     ! return value
```

status 对象也有其他的用途。函数 MPI_Recv() 的参数 source 和 tag 可以用专门的常量（“通配符”）MPI_ANY_SOURCE 和 MPI_ANY_TAG 分别填充。前者表明消息可以被任何人送，后者表明消息标签无关紧要。MPI_Recv() 返回后，status(MPI_SOURCE) 和 status(MPI_TAG) 分别包含了发送者的进程号及消息的标签。（C 语言中，status 对象属于 struct MPI_Status 类型的一部分，可以通过操作符“.”获得消息源和标签信息。）

注意，函数 MPI_Send() 和 MPI_Recv() 含有封闭的语义，这意味着在函数返回后缓冲区可以安全地使用（即 MPI_Send() 返回后对消息的更改不会对发送中的消息造成任何影响，同时，在函数 MPI_Recv() 返回后，也可以保证消息被完全接收了）。这一点不会和同步机制混淆，请看下面的例子。

代码清单 9-3 展示了一段并行在某个函数 f(x) 上计算积分的 MPI 程序段。与代码清单 6-2 中的 OpenMP 版本相反的是，MPI 中的各个进程间的任务分配必须手动来完成。根据进程号，每个 MPI 进程被分配了积分区域上相应的部分子区域（代码 9、10 行）。之后，函数 integrate() 可以在相应子区域上完成实际的积分（代码 13 行），这看上去和代码清单 6-2 很像。计算完成后，每个进程得到了自己的部分结果，这些结果需要累加在一起以得到最终的积分值。这个工作由进程 0 来完成，它在全部其他进程号 1 ～ size−1 之间执行循环（代码 18 ～ 29 行），依次调用函数 MPI_Recv(代码 19 行）从每个其他进程处接收各个部分积分值，计算最终结果并写回变量 res（代码 28 行）。除了进程 0，其余的所有进程均需调用 MPI_Send（ ）来传输数据。因此，总共有 size−1 个成对的接收和发送消息操作。收发两端的数据类型被指定为 MPI_DOUBLE_PRECISION, 也就是 Fortran 语言中通常的双精度类型（参见表 9-1 ）。这里没有使用消息标签，故我们将它设置为 0。

这个简单的程序可以通过下面的方式进行优化：

⊖ 对于 C/C++ 的 MPI 绑定，void* 指针类型可以很方便地隐藏参数类型的改变，因此不存在任何问题。但是这种改变对于 Fortran 的 MPI 绑定是明确不符合语言标准的，幸运的是在大多数情况下，这种改变都是可以容忍的。详情参见标准文档 [P15]。

- MPI标准并不保证消息传递的时序性，除了在相同的发送者、接收者（相同的标签）之间传递这种情况外。因此，为了使进程0能够无延迟地接收在不同执行时间下函数integrate()在其他进程上得到的结果，一种更好的方法是使用MPI_ANY_SOURCE通配符代替代码23行中确定的进程号。
- 进程0在自己调用的函数integrate（）返回前，没有调用函数MPI_Recv()。此时，如果其他进程先于进程0完成了自己那部分的计算，那么进程通信将被阻塞，而且
209 不能被计算所掩盖。MPI标准提供了非阻塞点对点通信设施，支持多个未完成的接收（和发送），甚至允许支持异步消息的实现。详见9.2.4节。

代码清单9-3 MPI并行积分程序片段

```
integer, dimension(MPI_STATUS_SIZE) :: status
call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)

! integration limits
a=0.d0 ; b=2.d0 ; res=0.d0

! limits for "me"
mya=a+rank*(b-a)/size
myb=mya+(b-a)/size

! integrate f(x) over my own chunk - actual work
psum = integrate(mya,myb)

! rank 0 collects partial results
if(rank.eq.0) then
   res=psum
   do i=1,size-1
      call MPI_Recv(tmp, &    ! receive buffer
                    1,   &    ! array length
                         ! data type
                    MPI_DOUBLE_PRECISION,&
                    i,   &    ! rank of source
                    0,   &    ! tag (unused here)
                    MPI_COMM_WORLD,& ! communicator
                    status,& ! status array (msg info)
                    ierror)
      res=res+tmp
   enddo
   write(*,*) 'Result: ',res
! ranks != 0 send their results to rank 0
else
   call MPI_Send(psum,    &    ! send buffer
                 1,       &    ! message length
                 MPI_DOUBLE_PRECISION,&
                 0,       &    ! rank of destination
                 0,       &    ! tag (unused here)
                 MPI_COMM_WORLD,ierror)
210 endif
```

- 由于进程0需要收集最终结果，当传递消息的数量增大时，该进程必然会产生通信瓶颈。在10.4.4节中，我们将会证明优化可以大幅度减少这种情况下的通信开销。幸运的是，没人需要为这一点写具体的代码。事实上，全局求和是归约操作的一个例子，并且它很好地被MPI所支持（参见9.2.3节）。目前厂商的实现已经默认提供这种全局操作的优化版本。

尽管MPI_Send()函数很容易使用，但在具体实现上，程序员应该意识到MPI标准给予

了它很大的自由。内在地，MPI_Send() 可能会完全同步地工作，这意味着在发出的信息尚未与接收者达成“握手”之前，函数调用不能返回到用户代码。然而，它也可能暂时将消息复制到中间缓冲区然后立即返回，让其他机制如后台线程完成握手以及数据传输。是否改变自己的行为取决于任何显式的或者隐藏的参数。如果没有考虑 MPI_Send() 执行时可能的同步性，那么除了对性能会造成影响，死锁亦有可能发生。一个典型的通信例子叫做“圆环转换”（参看图 9-1），所有进程位于闭合的圆环拓扑结构上，每个进程先向自己左边相邻的进程发送消息，然后从右边相邻的处理器接收消息：

```
1  integer :: size, rank, left, right, ierror
2  integer, dimension(N) :: buf
3  call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
4  call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)
5  left = rank+1                ! left and right neighbors
6  right = rank-1
7  if(right<0)     right=size-1   ! close the ring
8  if(left>=size) left=0
9  call MPI_Send(buf, N, MPI_INTEGER, left, 0, &
10                  MPI_COMM_WORLD,ierror)
11 call MPI_Recv(buf,N,MPI_INTEGER,right,0, &
12                  MPI_COMM_WORLD,status,ierror)
```

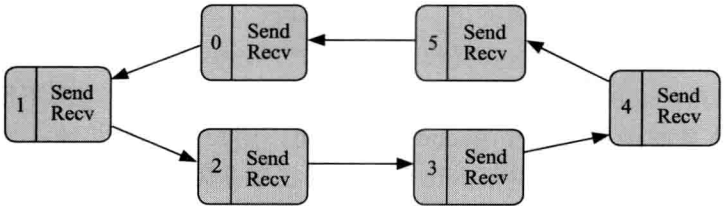


图 9-1 环形转换通信模式。假如发送和接收按图中的顺序执行，死锁将会发生，这是因为函数 MPI_Send() 可能是同步执行的 211

如果 MPI_Send() 是同步执行的，所有的进程先调用它，然后等待直到与之匹配的接收者发出接收信号。然而，如果通信的消息足够短小，环形转换也可能不会发生死锁。事实上，大多数的 MPI 实现版本提供了一个（小的）临时缓冲区来存放少量消息，当缓冲区填满或者太小的时候（这种情形实际上有一点复杂，详见 10.2 节和 10.3 节），MPI_Send() 才会切换到同步模式，这会导致不定时的死锁发生，而且很难检测到。如果你怀疑程序中的不定时死锁是由 MPI_Send() 切换到同步模式引起的，那么可以在程序中用 MPI_Ssend() 代替 MPI_send() 来检验，前者被定义成同步的并且与后者拥有同样的接口。

一种解决死锁问题的简答方法是互换 MPI_Send() 和 MPI_Recv() 调用的次序，比如调换这两个函数在所有奇数号进程的调用次序，这样对于每一个消息的发送都有一个与之匹配的接收（见图 9-2）。上述代码中 9 ～ 12 行应该被改写成：

```
1  if(MOD(rank,2)/=0) then
2    call MPI_Recv(buf,N,MPI_INTEGER,right,0, &     ! odd rank
3                  MPI_COMM_WORLD,status,ierror)
4    call MPI_Send(buf, N, MPI_INTEGER, left, 0, &
5                  MPI_COMM_WORLD,ierror)
6  else
7    call MPI_Send(buf, N, MPI_INTEGER, left, 0, & ! even rank
8                  MPI_COMM_WORLD,ierror)
9    call MPI_Recv(buf,N,MPI_INTEGER,right,0, &
10                 MPI_COMM_WORLD,status,ierror)
11 endif
```

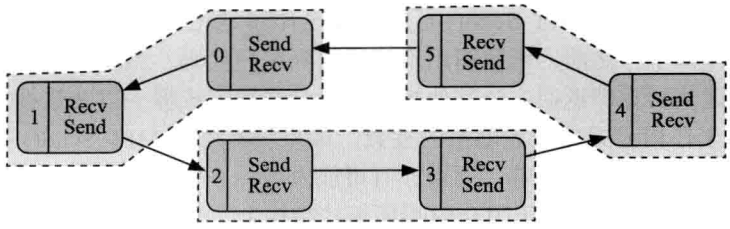


图 9-2 一种环形转换死锁问题的可能解法：改变奇数号进程上 MPI_Send() 和 MPI_Recv() 的顺序，这样对于每一个发送消息的操作都有一个与之相匹配的接收操作（虚线框内），因此成对的进程可以避免死锁而进行通信

在一次由偶数进程号传递的消息结束后，剩下的发送 / 接收对也能够进行匹配。这种方法没有充分利用非阻塞网络的全部带宽，因为在任何时刻仅有一半的通信连接是活跃的（当 MPI_Send() 是完全同步的）。更好的替代方法是使用非阻塞通信。详细内容参见 9.2.4 节及
212 习题 9.1 以更多地了解环形转换模式。

由于环形转换或与其相似的模式是普遍存在的，即使在阻塞通信的情况下，MPI 对它们也有一些直接的支持。函数 MPI_Sendrecv() 和 MPI_Sendrecv_replace() 将标准的发送和接收结合在一次调用当中。其中后者使用一个通信缓冲区，接收的消息会覆盖发送的数据。这两个函数都能确保不会如单独发送和接收消息那样产生死锁。

最后需要补充一点的是，MPI 中也有在不考虑接收者状态的情况下能返回到用户代码的阻塞发送消息的函数（MPI_Bsend()）。然而，用户必须显式地在发送端提供足够的缓冲空间。这种方法在实际中很少出现，原因在于非阻塞通信相对更容易使用（参见 9.2.4 节）。

9.2.3 集合通信

如上文展示的那样将各部分计算结果进行收集是归约操作的一个实例，它执行在通信空间中的所有进程上。在 OpenMP 的相关章节中，我们已经对归约进行了介绍（见 6.1.5 节），它们有着相同的目的。在 MPI 中，也有使归约更简单的机制，这种机制在大多数情况下相比于在所有的进程号间循环来收集结果的方式更高效。由于归约是在同一通信空间中的所有进程共同参与的一个过程，因此在 MPI 中，它属于所谓的集合 (collective) 或者全局通信（global communication）操作。不同于点对点通信，集合通信需要每个进程调用相同的函数，因此对于一个点对点的消息发送，如 MPI_Send()，是不可能匹配使用集合方式初始化的接收者的。

MPI 中最简单的集合是栅栏（barrier），它不做任何实际上的数据转换：

```
integer :: comm, ierror
call MPI_Barrier(comm,    ! communicator
                 ierror)  ! return value
```

栅栏使同一通信空间中的所有成员同步，即在返回各自的用户代码之前，所有的进程必须调用这个函数。尽管最初的时候人们频繁地使用它，但是在 MPI 中它的重要性通常是被高估的，因为其他的 MPI 例程考虑到了用更好的控制来实现隐式或者显式的同步。尽管如此，它有时也被用来调试和分析程序。

广播是一个更有用的集合，它从一个进程（“根”）发送消息到通信空间中的所有其他进程中：

```
<type> buf(*)
integer :: count, datatype, root, comm, ierror
call MPI_Bcast(buffer,     ! send/receive buffer
               count,      ! message length
               datatype,   ! MPI data type
               root,       ! rank of root process
               comm,       ! communicator
               ierror)     ! return value
```

213

“根”是一些通用数据源所在的地方，这个概念对于很多集合例程来说都是相通的。尽管将 0 号进程选为“根”是很自然的，但是它并不与其他进程不同。MPI_Bcast() 的缓冲区参数是位于“根”上的发送缓冲区和任意其他进程上的接收缓冲区（见图 9-3）。正如已经提到的那样，通信空间中的每个进程必须调用这个历程，当然所有其他调用的 root 参数必须相同。当一个进程含有的信息必须同其他进程共享时需要使用广播。例如，在程序开始运行后，通常会有一个执行一些初始化操作的进程，如读参数文件或者命令行参数。随后，它就可以将这些读到的数据通过 MPI_Bcast() 与其他进程通信。

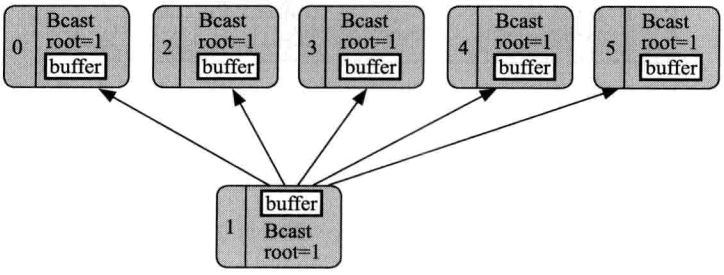


图 9-3 一个 MPI 广播：“根”进程（本例中标号为 1）向所有其他进程发送相同的消息。通信空间中的每个节点必须使用相同的根参数调用 MPI_Bcast()

许多更高级的集合调用通常与全局数据分布有关。MPI_Gather() 收集所有进程发送缓冲区的内容，以一定的次序将它们连接然后写入根进程的接收缓冲区。MPI_Scatter() 执行转置，它将根进程的发送缓冲区分成相同大小的块。它们存在于支持任意块大小的变量中（在名字上附加“v”）。MPI_Allgather() 是 MPI_Gather() 和 MPI_Bcast() 的结合。表 9-3 给出了更多的例子。

回到之前求积分的那个例子，我们已经表明存在更高效的方法计算全局归约，这就是函数 MPI_Reduce():

```
<type> sendbuf(*), recvbuf(*)
integer :: count, datatype, op, root, comm, ierror
call MPI_Reduce(sendbuf,     ! send buffer
                recvbuf,     ! receive buffer
                count,       ! number of elements
                datatype,    ! MPI data type
                op,          ! MPI reduction operator
                root,        ! root rank
                comm,        ! communicator
                ierror)      ! return value
```

MPI_Reduce() 根据 op 参数指定的操作类型，将所有进程上 sendbuf 数组中的内容按元素位置对应结合起来并将结果存储在根进程的 recvbuf 中（参见图 9-4）。共有 12 种预定义的操作，其中最重要的 4 种是 MPI_MAX、MPI_MIN、MPI_SUM 和 MPI_PROD，它们分别实现了计算全局最大值、最小值、求和以及乘积。它同时也支持用户自定义的操作。

214

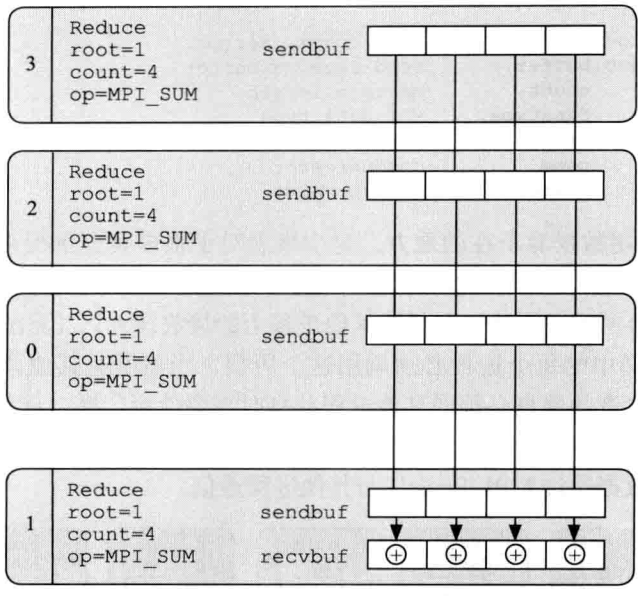


图 9-4 一个用 MPI_Reduce() 实现的长度为 count 的数组归约（本例中是求和）。每个进程必须提供一个发送缓冲区，接收缓冲区参数仅使用在根进程上。根进程上的局部复制可以通过用 MPI_IN_PLACE 代替发送缓冲区地址来避免

现在已经很清楚了，代码清单 9-3 中第 16 ～ 39 行的整个 if…else…endif 结构（除了第 30 行打印结果）可以改写如下：

```
1 call MPI_Reduce(psum, &          ! send buffer (partial result)
2                 res,  &          ! recv buffer (final result @ root)
3                 1, &             ! array length
4                 MPI_DOUBLE_PRECISION, &
5                 MPI_SUM, &       ! type of operation
6                 0, &             ! root (accumulate result there)
7                 MPI_COMM_WORLD,ierror)
```

尽管接收缓冲区（这里是 res 变量）必须在所有进程上指定，它却仅使用在根进程上。通常情况下，MPI_Reduce() 在根进程上需要独立的发送和接收缓冲区。如果程序语义允许，那么在根进程上的本地运算可以通过将 sendbuf 参数设置成固定的常量 MPI_IN_PLACE 来简化。recvbuf 被用作发送缓冲区，被全局结果所覆盖。当 count 值很大，并且多余的复制操作会导致明显的开销时，这种方式对性能很有帮助。在其他所有的非根进程上调用的方式并不改变。

215 还有一些其他关于 MPI_Reduce() 的全局操作值得注意。例如，MPI_Allreduce() 是一种归约和广播的融合；MPI_Reduce_scatter() 结合了函数 MPI_Reduce() 及 MPI_Scatter()。

集合体不需要同步操作就能使所有进程同步（当然，栅栏在定义上是同步的），因此可能诱发和阻塞点对点通信相似的死锁危险（见上面）。这意味着，集合体必须在所有进程上以相同的顺序执行。参见 MPI 标准文档 [P15] 中的例子。

通常情况下，相比于点对点的结构或者“模仿”同样语义的简约集合体的组合，使用集合体是一个好的想法（参见图 10-15 和图 10-16 以及 10.4.4 节的相关讨论）。好的 MPI 实现版本为集合通信上的数据流做了优化，同时也内置了一些网络的拓扑结构信息。

9.2.4 非阻塞点对点通信

目前为止所有提到的 MPI 函数都有一个相同的属性，即函数调用只有在消息传递得足够远时才会返回，这样保证了发送 / 接收缓冲区可以无顾虑地使用。这意味着，接收数据已经完全到达，发送数据也完全离开了缓冲区，因此可以安全地改动缓冲区而不对消息造成影响。这在 MPI 术语中称作阻塞通信。尽管在当前的 MPI 标准（版本 2.2）中，MPI 的集合通信总是阻塞的，但是点对点通信是可以使用非阻塞语义执行的。非阻塞点对点调用仅初始化了消息传输，然后立即返回到用户代码。在一个高效的实现版本中，等待数据到达和实际的数据传递在后台进行，让资源自由计算，同步被移除（图 9-5 展示了非阻塞调用 MPI_Isend() 的一种可能的事件时间轴）。换言之，非阻塞 MPI 是一种在高效实现下，通信和计算可以异步执行的方式。只要用户程序没被告知这样做是安全的（可以用适当的 MPI 调用检验），消息缓冲区就不能被使用。非阻塞和阻塞 MPI 调用是相互兼容的，即通过阻塞调用发送的消息能够被非阻塞接收所匹配。 216

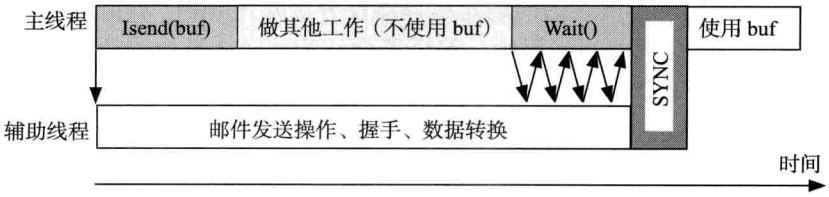


图 9-5 非阻塞发送（MPI_Isend()）的抽象时间轴图。标准没有指定是否有一个辅助线程，全部的数据转换将在 MPI_Wait() 或任何其他 MPI 函数期间执行

最重要的非阻塞通信是 MPI_Isend():

```
<type> buf(*)
integer :: count, datatype, dest, tag, comm, request, ierror
call MPI_Isend(buf,          ! message buffer
               count,        ! # of items
               datatype,     ! MPI data type
               dest,         ! destination rank
               tag,          ! message tag
               comm,         ! communicator
               request,      ! request handle (MPI_Request* in C)
               ierror)       ! return value
```

与阻塞发送相反，MPI_Isend() 有一个额外的输出参数，即*请求句柄*。它起到标识符的作用，程序可以随后通过它指定“等待”通信的请求（在 C 语言中，它属于结构体 MPI_Request 类型）。相应地，MPI_Irecv() 初始化一个非阻塞接收：

```
<type> buf(*)
integer :: count, datatype, source, tag, comm, request, ierror
call MPI_Irecv(buf,          ! message buffer
               count,        ! # of items
               datatype,     ! MPI data type
               source,       ! source rank
               tag,          ! message tag
               comm,         ! communicator
               request,      ! request handle
               ierror)       ! return value
```

这里不含有 MPI_Recv() 中存在的参数对象 status，因为并不需要，毕竟当函数调用返回到用户代码的时候，并没有实际的通信发生。可以通过函数 MPI_Test() 和 MPI_Wait() 来检验一个正在进行中的通信是否完成。前者仅检验通信是否完成并返回一个标识，后者将缓

冲区阻塞直到可以使用。

```
1 logical :: flag
2 integer :: request, status(MPI_STATUS_SIZE), ierror
3 call MPI_Test(request,      ! pending request handle
4               flag,         ! true if request complete (int* in C)
5               status,       ! status object
6               ierror)       ! return value
7 call MPI_Wait(request,      ! pending request handle
8               status,       ! status object
9               ierror)       ! return value
```

仅当进行中的通信是一个完全被接收的消息参数对象 status 才会包含有用的信息（例如，在 MPI_Test() 的例子中，flag 的值必须是 true）。在这种意义上，序列

```
1 call MPI_Irecv(buf, count, datatype, source, tag, comm, &
2                request, ierror)
217 3 call MPI_Wait(request, status, ierror)
```

完全等价于标准的 MPI_Recv()。

非阻塞 MPI 通信的一个潜在的问题是编译器没办法知道 MPI_Wait() 可以（通常会）改变 buf 中的内容。因此，下面的代码中，编译器会认为将第 3 行中的语句移到 MPI_Wait() 调用之前是合法的：

```
1 call MPI_Irecv(buf, ..., request, ...)
2 call MPI_Wait(request, status, ...)
3 buf(1) = buf(1) + 1
```

这样会导致竞争条件，同时 buf 中的内容可能是错的。通过请求句柄处理的 MPI_Irecv() 和 MPI_Wait() 之间的内在联系对编译器是不可见的，事实上，buf 没有包含在 MPI_Wait() 的参数列表中就足以认为代码修改是合法的。避免这种情况的一个简单的方法是把变量（或者缓冲区）放入一个 COMMON 块中，这样所有的子程序可以潜在地修改它。参见 MPI 标准 [P15] 作为备选。

多重请求可以在任何时候等待，这是非阻塞通信的又一个巨大优势。有时候，一组请求在一些方面可以放在一起，用户想要的检验可能不是指定的一个，而可能是任意一个或者任意数量，甚至它们中的全部是否完成。这些可以通过适当地使用句柄数组参数化的函数调用来检验。我们选用 MPI_Waitall() 程序来作为例子：

```
1 integer :: count, requests(*)
2 integer :: statuses(MPI_STATUS_SIZE,*), ierror
3 call MPI_Waitall(count,          ! number of requests
4                  requests,       ! request handle array
5                  statuses,       ! statuses array (MPI_Status* in C)
6                  ierror)         ! return value
```

函数调用仅在所有进行中的请求全都完成后才会返回。status 对象在 array_of_statuses(:,:) 中可以使用。

代码清单 9-3 中的例子可以通过重叠进程 0 上的局部积分和接收其他进程的结果来使用非阻塞通信。不幸的是，此处不能使用集合体，因为在 MPI 中没有非阻塞集合体。代码清单 9-4 中展示了一个可能的结果。在原始代码中，归约操作不得不手动完成（代码 33 ～ 35 行）。在编译时，状态和请求数组的大小是未知的，因此和除了 0 号进程的所有其他独立接收缓冲区（代码 11 ～ 13）一样必须动态分配。部分结果的收集是通过代码 32 行的单独 MPI_Waitall() 执行的。MPI_Send() 足以传输部分结果（代码 39 行）而不需要在非根进程上

做任何改动。

非阻塞通信提供了一个明显的方式（如花销）来重叠有用工作之间的通信，即开销。然而，可能产生的性能优势取决于许多因素，甚至可能已不复存在（参见 10.4.3 节）。但是即使没有真正的重叠，多重显著的非阻塞请求也会提升性能，这是因为 MPI 库会决定它们中的哪些会最先得到服务。 218

代码清单 9-4 MPI 并行积分程序片段，使用非阻塞点对点通信

```
integer, allocatable, dimension(:,:) :: statuses
integer, allocatable, dimension(:) :: requests
double precision, allocatable, dimension(:) :: tmp
call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)

! integration limits
a=0.d0 ; b=2.d0 ; res=0.d0

if(rank.eq.0) then
  allocate(statuses(MPI_STATUS_SIZE, size-1))
  allocate(requests(size-1))
  allocate(tmp(size-1))
! pre-post nonblocking receives
  do i=1,size-1
    call MPI_Irecv(tmp(i), 1, MPI_DOUBLE_PRECISION, &
                   i, 0, MPI_COMM_WORLD,        &
                   requests(i), ierror)
  enddo
endif

! limits for "me"
mya=a+rank*(b-a)/size
myb=mya+(b-a)/size

! integrate f(x) over my own chunk - actual work
psum = integrate(mya,myb)

! rank 0 collects partial results
if(rank.eq.0) then
   res=psum
   call MPI_Waitall(size-1, requests, statuses, ierror)
   do i=1,size-1
      res=res+tmp(i)
   enddo
   write (*,*) 'Result: ',res
! ranks != 0 send their results to rank 0
else
   call MPI_Send(psum, 1, &
                 MPI_DOUBLE_PRECISION, 0, 0, &
                 MPI_COMM_WORLD,ierror)
endif
```

219

表 9-3 给出了在 MPI 中可用的通信模式以及最重要的库函数的一个概述。

表 9-3 MPI 通信模式及相关子程序的不全面概述

	点对点	集合
阻塞	MPI_Send() NPI_Ssend() MPI_Bsend() MPI_Recv()	MPI_Barrier() MPI_Bcast() MPI_Scatter()/ MPI_Gather() MPI_Reduce() MPI_Reduce_scatter() MPI_Allreduce()

（续）

	点对点	集合
非阻塞	MPI_Isend() MPI_Irecv() MPI_Wait()/MPI_Test() MPI_Waitany()/ MPI_Testany() MPI_Waitsome()/ MPI_Testsome() MPI_Waitall()/ MPI_Testall()	N/A

9.2.5 虚拟拓扑

在5.2.1节中，我们指出了区域分解的原则作为数据并行的实例。使用目前为止提到的MPI函数完全能够在分布式存储的并行计算机上实现区域分解。然而，建立处理进程网格以及跟踪哪个进程需要交换边缘数据却是不容易的。由于区域分解方法的重要性，因此MPI包含了一些函数来以*虚拟拓扑*的形式支持这个周期性任务。虚拟拓扑提供了一个方便的进程命名方案，并且符合需要的通信模式。此外，它们潜在地允许MPI库通过应用网络拓扑的知识来优化通信。尽管可以用MPI来描述任意的图拓扑结构，但此处我们只考虑笛卡儿拓扑。

作为一个例子，假定有一个仿真需要处理一个包含 $3000 \times 4000 = 1.2 \times 10^7$ 个数据的大型双精度数组 $P(1:3000,1:4000)$。仿真运行在 $3 \times 4 = 12$ 个进程上，数组在它们之上“自然”分布，即每个进程持有一个 1000×1000 大小的数据块。图9-6展示了反映这种情况的一种可
220 能的笛卡儿拓扑：每个进程可以被它的进程号或者笛卡儿坐标所标识。它有很多的邻居节点，这些节点的数目由网格的维度决定。在我们的例子中维度为2，这导致每个进程最多有4个相邻节点。每个维度上的边界条件可以是闭（环）的或者开的。

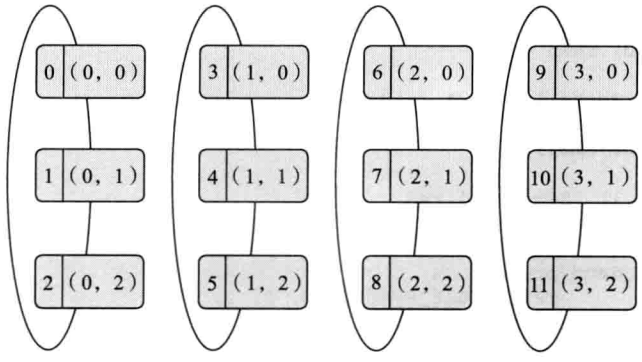


图9-6 二维笛卡儿拓扑：12个进程分布在一个 3×4 的网格上，且在网格的第二维上是周期性的，第一维度上并不是。图上标注了MPI进程号以及笛卡儿坐标

MPI能够帮助在处理器网格中建立进程号和笛卡儿坐标之间的映射关系。首先，新的通信空间必须定义成选定的拓扑结构附着的地方。这通过函数MPI_Cart_create()来实现：

```
integer :: comm_old, ndims, dims(*), comm_cart, ierror
logical :: periods(*), reorder
call MPI_Cart_create(comm_old,      ! input communicator
```

```
                    ndims,          ! number of dimensions
                    dims,           ! # of processes in each dim.
                    periods,        ! periodicity per dimension
                    reorder,        ! true = allow rank reordering
                    comm_cart,      ! new cartesian communicator
                    ierror)         ! return value
```

它产生一个新的“笛卡儿”通信空间 comm_cart，稍后将被用来指代特定的拓扑结构。periods 数组指定了笛卡儿坐标方向是周期的；reorder 参数为真时，将允许进程号重排，这样进程号在通信空间 comm_old 和 comm_cart 中将会不同。MPI 库将通过使用它关于网络拓扑结构的知识选择一个不同顺序，同时期望下一个通信在笛卡儿拓扑结构中是显性的。当然，任意两个进程在笛卡儿通信空间中的通信仍然是允许的。

这里并没有提到实际问题的大小（3000 × 4000），因为关心数据的分布完全是用户的工作。MPI 能做的事情就是记录拓扑的信息。对于图 9-6 中的拓扑结构，MPI_Cart_create() 可以调用如下：

```
call MPI_Cart_create(MPI_COMM_WORLD,          ! standard communicator
                     2,                       ! two dimensions
                     (/ 4, 3 /),              ! 4x3 grid
                     (/ .false., .true. /),   ! open/periodic
                     .false.,                 ! no rank reordering
                     comm_cart,               ! Cartesian communicator
                     ierror)
```

221

如果 MPI 进程的数量给定，那么找到每个方向上网格的最佳扩展（正如在 MPI_Cart_create() 中参数 dims 需要的那样）需要一些算数运算，这些具体数据可以被卸载到 MPI_Dims_create() 函数中：

```
integer :: nnodes, ndims, dims(*), ierror
call MPI_Dims_create(nnodes, ! number of nodes in grid
                     ndims,  ! number of Cartesian dimensions
                     dims,   ! input: /=0 # nodes fixed in this dir.
                             !        ==0 # calculate # nodes
                             ! output: number of nodes each dir.
                     ierror)
```

dims 数组既作为输入参数又作为输出参数：dims 中的每个输入相当于一个笛卡儿维度，输入为 0 表示这个维度上 MPI_Dims_create() 需要计算进程的数量；输入非 0 则用于指定这个维度上进程的固定数量。在上述约束下，函数决定了一个平衡的分布，即所有的 ndims 扩展尽可能近的挨在一起。考虑到通信的开销，只有当整体问题的网格是立方体的时候才是最佳的分布。如果情况并非如此，则由用户负责设定合适的约束，这是因为 MPI 没办法获知网格的几何形状。

MPI 提供了两个服务函数用来实现在笛卡儿进程坐标和 MPI 进程号之间的转换。对于给定的进程号，MPI_Cart_coords() 函数计算相应的笛卡儿进程坐标：

```
integer :: comm_cart, rank, maxdims, coords(*), ierror
call MPI_Cart_coords(comm_cart,     ! Cartesian communicator
                     rank,          ! process rank in comm_cart
                     maxdims,       ! length of coords array
                     coords,        ! return Cartesian coordinates
                     ierror)
```

（如果在产生 comm_cart 时进程号重排是允许的，则进程应该总是先调用 MPI_Comm_rank(comm_cart,…) 以获得它的进程号）。输出数组 coords 包含了对应于指定进程的笛卡儿坐标。

当涉及区域分解的时候，映射函数总是需要的。一个进程从 MPI 中获得的第一个信息就是它在笛卡儿空间中的进程号。这就需要 MPI_Cart_coords() 来决定坐标以便进程能够进行计算，例如，进程应该工作在哪个子区域。具体实例参见 9.3 节。

反向映射，即从笛卡儿坐标到 MPI 进程号，通过 MPI_Cart_rank() 实现：

```
integer :: comm_cart, coords(*), rank, ierror
call MPI_Cart_rank(comm_cart,     ! Cartesian communicator
                   coords,        ! Cartesian process coordinates
                   rank,          ! return process rank in comm_cart
                   ierror)
```

222

一个典型的区域分解任务是找出在特定的笛卡儿维度上给定进程的相邻进程。原则上，用户可以从笛卡儿坐标开始，通过调用 MPI_Cart_rank() 一个接一个地（考虑开启或者关闭边界条件）将结果映射到 MPI 进程号。所有这些任务，MPI_Cart_shift() 可以一步完成：

```
integer :: comm_cart, direction, disp, rank_source,
integer :: rank_dest, ierror
call MPI_Cart_shift(comm_cart,    ! Cartesian communicator
                    direction,    ! direction of shift (0..ndims-1)
                    disp,         ! displacement
                    rank_source, ! return source rank
                    rank_dest,    ! return destination rank
                    ierror)
```

参数 direction 指定转换应该执行在哪个笛卡儿维度上，disp 决定距离和方向（正或负）。rank_source 和 rank_dest 返回由其他参数确定的相邻进程号。图 9-7 展示了一个在第一维度上负方向的转换，它执行在图 9-6 中给定拓扑结构的进程 4 上。源和目标“邻居”的进程号分别是 7 和 1。如果由于相邻节点的扩展超过了在非环形维度上的网格边界而导致“邻居”不存在，则进程会返回特殊值 MPI_PROC_NULL，使用 MPI_PROC_NULL 作为源或者目标进程号在任何通信调用中都是允许的，它会对这个调用反馈一个空语句，即不会发生任何实际的通信。这样可以使编程变得简单，因为网格的边界不需要作为特殊情况来专门对待（参见 9.3 节中的例子）。

223

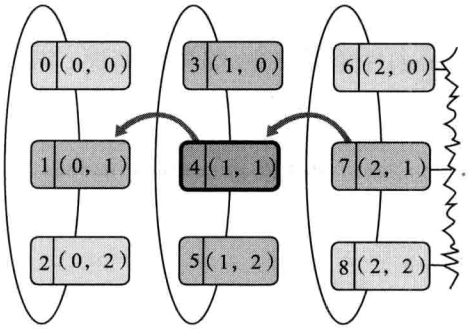


图 9-7 MPI_Cart_shift() 作用在图 9-6 一部分拓扑结构中的结果。通过进程 4 来执行，direction=0，disp=−1，函数返回 rank_source=7 及 rank_dest=1

9.3 实例：Jacobi 解法器的 MPI 并行

我们用一个简单的三维 Jacobi 解法器（见 3.3 节和 6.2 节）作为一个虚拟拓扑和其他 MPI 函数的非对称例子。与使用插入一组指导语句以实现并行化的 OpenMP 相反，使用区域分解的 MPI 并行化更加复杂。

9.3.1 MPI 实现

尽管在 5.2.1 节中我们已经给出了基本算法，但这里我们仍需要更多的细节。图 9-8 展示了一个注解流程图。中心部分仍是扫描所有的子区域（步骤 3），这是主要的计算部分。然而每个子区域被不同的 MPI 进程所处理，因而面临着两个困难：

1）收敛条件依赖于所有网格单元的当前以及下一个时间步长之间的最大偏差。对于每

个子区域，这个值很容易分别得到，但是归约需要得到一个全局的最大值。

2）为了在子区域上扫描产生正确的结果，必须实现恰当的边界条件。这对于位于实际边界附近的网格单元是没有问题的，但是对于毗邻区域分割处的网格单元，边界条件随着一次次扫描而改变：这个边界被位于分割边界右边的单元所包括，而这些单元由于被别的 MPI 进程拥有而不能直接访问。（在 OpenMP 中，所有数据总是对全部线程可见，这使得跨“块边界”的访问变得微不足道。）

在每个进程已经在自己的区域内获得了最大的偏差 maxdelta 之后，第一个问题通过 MPI_Allreduce() 调用直接解决了（图 9-8 中的步骤 4）。

至于第二个问题，所谓的 ghost 或者 halo 层被用来存储相邻区域的边界信息的副本。由于对于每次区域分解，每个子区域仅需要一个 ghost 层，故不需要申请额外的内存，这是因为可以利用存储边界层的空间。（然而下面我们将看到，由于一些技巧的原因可能会需要一些额外的数组。）在执行从 T=0 到 T=1 时刻数据更新的进程扫描自己的子区域之前，T=0 时刻进程通过 MPI 从它的相邻单元获得边界值并将其存储在 ghost 单元中（图 9-8 中的步骤 2）。下面我们将会概述用 Fortran 实现这个算法的核心部分。完整的代码可以从本书的网站上下载[⊖]。为清楚起见，我们将会使用代码片段来阐明一些重要的变量。 224

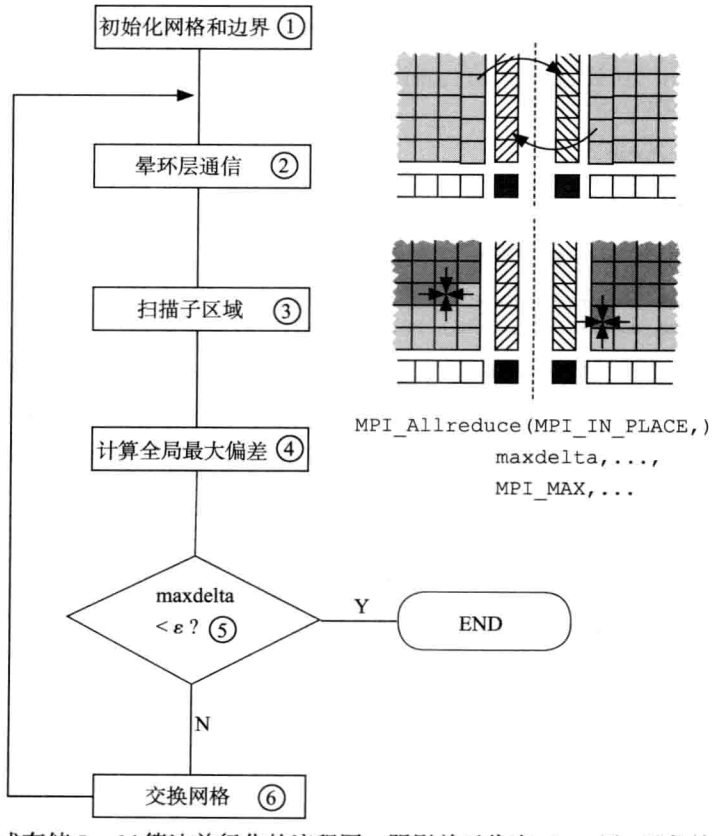


图 9-8 分布式存储 Jacobi 算法并行化的流程图，阴影单元代表 ghost 层，黑色单元代表在 T=1 时刻的网格已经更新过，浅色单元代表 T=0 时刻的数据。白色单元代表整体网格上的真实边界，黑色单元表示未被使用的边界 225

⊖ http://www.hpc.rrze.uni-erlangen.de/HPC4SE/。

首先0号进程从标准的输入读入所需的参数（下述代码的第10行）：在所有维度上问题的大小（spat_dim）、可能使用的进程数目的预置（proc_dim）以及频率（pbc_check）。

```
1  logical, dimension(1:3) :: pbc_check
2  integer, dimension(1:3) :: spat_dim, proc_dim
3
4  call MPI_Comm_rank(MPI_COMM_WORLD, myid, ierr)
5  call MPI_Comm_size(MPI_COMM_WORLD, numprocs, ierr)
6
7  if(myid.eq.0) then
8    write(*,*) ' spat_dim , proc_dim, PBC ? '
9    do i=1,3
10     read(*,*) spat_dim(i), proc_dim(i), pbc_check(i)
11   enddo
12 endif
13
14 call MPI_Bcast(spat_dim , 3, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
15 call MPI_Bcast(proc_dim , 3, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
16 call MPI_Bcast(pbc_check, 3, MPI_LOGICAL, 0, MPI_COMM_WORLD, ierr)
```

尽管许多的MPI实现版本有选项来允许进程0的标准输入对所有进程可见，但一个可移植的MPI程序不能依赖于这个特征，因而进程0必须将这些数据广播出去（代码第14～16行）。之后，笛卡儿拓扑才能够使用MPI_Dims_create()和MPI_Cart_create()来建立：

```
1  call MPI_Dims_create(numprocs, 3, proc_dim, ierr)
2
3  if(myid.eq.0) write(*,'(a,3(i3,x))') 'Grid: ', &
4      (proc_dim(i),i=1,3)
5
6  l_reorder = .true.
7  call MPI_Cart_create(MPI_COMM_WORLD, 3, proc_dim, pbc_check, &
8      l_reorder, GRID_COMM_WORLD, ierr)
9
10 if(GRID_COMM_WORLD .eq. MPI_COMM_NULL) goto 999
11
12 call MPI_Comm_rank(GRID_COMM_WORLD, myid_grid, ierr)
13 call MPI_Comm_size(GRID_COMM_WORLD, nump_grid, ierr)
```

因为允许进程重排（代码第6行），必须再次调用MPI_Comm_rank()重新获得进程号（代码第12行）。此外，新的笛卡儿通信空间GRID_COMM_WORLD可能会比MPI_COMM_WORLD要小。余下的进程接收通信器值MPI_COMM_NULL，同时被发送到一个栅栏以等待全部的并行程序完成（代码第10行）。

现在拓扑结构已经建立完成，局部子区域可以被建立，包括内存申请：

```
1  integer, dimension(1:3) :: loca_dim, mycoord
2
3  call MPI_Cart_coords(GRID_COMM_WORLD, myid_grid, 3,
4      mycoord,ierr)
5
6  do i=1,3
7    loca_dim(i) = spat_dim(i)/proc_dim(i)
8    if(mycoord(i) < mod(spat_dim(i),proc_dim(i))) then
9      local_dim(i) = loca_dim(i)+1
10   endif
11 enddo
12
13 iStart = 0 ; iEnd = loca_dim(3)+1
14 jStart = 0 ; jEnd = loca_dim(2)+1
15 kStart = 0 ; kEnd = loca_dim(1)+1
16
17 allocate(phi(iStart:iEnd, jStart:jEnd, kStart:kEnd,0:1))
```

226

使用数组 mycoord 存储代码 3 行中调用 MPI_Cart_coords() 获得的进程笛卡儿坐标。数组 loca_dim 维护进程子区域在三维空间的扩展。这些数值由代码 6 ～ 11 行计算得出。代码 17 行执行内存申请，在所有的方向考虑了一个额外的层，该层用于在需要时存储边界值或者晕环值。为了简便起见，这里我们省略了数组的初始化以及外层网格边界。

用于 ghost 层交换的点对点通信需要连续的消息缓冲区。(事实上，这里会有一个选项用于选择使用导出的 MPI 数据类型，但这将超出本书的介绍范围。) 然而，只有那些在中间维度 (i) 连续的边界单元在内存中也是连续分布的。在任意 i–j、i–k 及 j–k 平面上的完整的层均是不连续的，因此必须使用一个中间缓冲区收集需要通信到相邻 ghost 层的边界数据。发送每个连续的块作为独立的消息是毫无可能的，因为这样会使网络溢出短消息，同时每个请求的延迟也要被容忍 (关于优化 MPI 通信的更多信息详见第 10 章)。

在每个进程上，我们使用两个中间缓冲区，一个用于发送，另一个用于接收。由于晕环数据的数量在不同的笛卡儿方向上可以是不同的，因此中间缓冲区的大小必须选取为可以容纳可能产生的最大晕环：

```
1  integer, dimension(1:3) :: totmsgsize
2
3  ! j-k plane
4  totmsgsize(3) = loca_dim(1)*loca_dim(2)
5  MaxBufLen=max(MaxBufLen,totmsgsize(3))
6  ! i-k plane
7  totmsgsize(2) = loca_dim(1)*loca_dim(3)
8  MaxBufLen=max(MaxBufLen,totmsgsize(2))
9  ! i-j plane
10 totmsgsize(1) = loca_dim(2)*loca_dim(3)
11 MaxBufLen=max(MaxBufLen,totmsgsize(1))
12
13 allocate(fieldSend(1:MaxBufLen))
14 allocate(fieldRecv(1:MaxBufLen))
```

227

与此同时，三个方向上的晕环大小被存储在了整数数组 totmsgsize 中。

现在，我们开始实现主要的迭代循环，循环的次数为最大的迭代 (扫描) 数 ITERMAX：

```
1  t0=0 ; t1=1
2  tag = 0
3  do iter = 1, ITERMAX
4    do disp = -1, 1, 2
5      do dir = 1, 3
6
7        call MPI_Cart_shift(GRID_COMM_WORLD, (dir-1), &
8                            disp, source, dest, ierr)
9
10       if(source /= MPI_PROC_NULL) then
11         call MPI_Irecv(fieldRecv(1), totmsgsize(dir), &
12              MPI_DOUBLE_PRECISION, source, &
13              tag, GRID_COMM_WORLD, req(1), ierr)
14       endif    ! source exists
15
16       if(dest /= MPI_PROC_NULL) then
17         call CopySendBuf(phi(iStart, jStart, kStart, t0), &
18                          iStart, iEnd, jStart, jEnd, kStart, kEnd, &
19                          disp, dir, fieldSend, MaxBufLen)
20
21         call MPI_Send(fieldSend(1), totmsgsize(dir), &
22                       MPI_DOUBLE_PRECISION, dest, tag, &
23                       GRID_COMM_WORLD, ierr)
24       endif    ! destination exists
25
```

```
26        if(source /= MPI_PROC_NULL) then
27          call MPI_Wait(req, status, ierr)
28
29          call CopyRecvBuf(phi(iStart, jStart, kStart, t0), &
30                           iStart, iEnd, jStart, jEnd, kStart, kEnd, &
31                           disp, dir, fieldRecv, MaxBufLen)
32        endif    ! source exists
33
34      enddo    ! dir
35    enddo    ! disp
36
37    call Jacobi_sweep(loca_dim(1), loca_dim(2), loca_dim(3), &
38                      phi(iStart, jStart, kStart, 0), t0, t1, &
39                      maxdelta)
40
41    call MPI_Allreduce(MPI_IN_PLACE, maxdelta, 1, &
42                       MPI_DOUBLE_PRECISION, &
43                       MPI_MAX, 0, GRID_COMM_WORLD, ierr)
44    if(maxdelta<eps) exit
45    tmp=t0; t0=t1; t1=tmp
46  enddo    ! iter
47
48  999 continue
```

晕环的交换分成6步完成，如各自在每个正、负笛卡儿方向上。通过循环变量disp和
228 dir实现参数化。代码7行，调用MPI_Cart_shift()来决定在当前方向（源和目的）上的通信邻居。如果子区域位于网格的边界，同时周期边界条件未知，则邻居节点将被告知进程号为MPI_PROC_NULL。使用这个进程号作为源或者目的的MPI调用将会立即返回。然而在这个例子中为了效率起见，复制到中间缓冲区以及从中间缓冲区复制应该被避免。我们屏蔽了任何的MPI调用，保证通信开销最低（代码10、16及26行）。

沿着一个方向上的通信模式实际上是一个环形转换（或者开边界条件下的线性转换）。本质上是一个阻塞点对点通信的环形转换在9.2.2节中已经讨论过。为了避免死锁同时尽可能地利用已有的网络带宽，在其他任何事情开始前，先初始化一个非阻塞接收（代码11行）。这个数据传输可以潜在地掩盖随后通过调用子程序CopySendBuf()完成的到中间发送缓冲区的晕环数据拷贝（代码17行）。在发送晕环数据（代码21行）以及等待之前的非阻塞接收完成后（代码27行），CopyRecvBuf()最后将接收的晕环数据拷贝到边界层（代码29行），从而完成了在一个特定方向上的通信环。图9-9再一次阐明了整个过程。

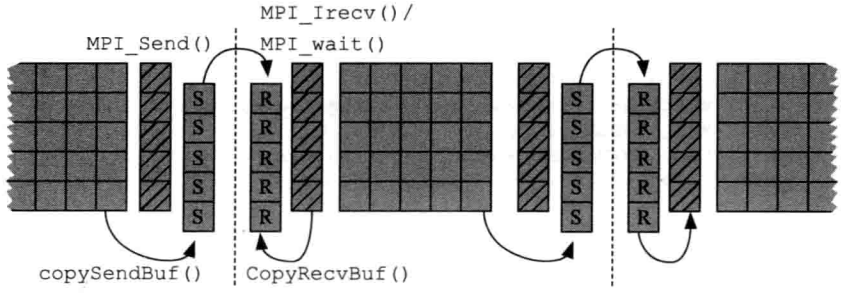


图9-9 Jacobi解法器沿着一个笛卡儿方向的晕环通信（图例展示的是二维情况）。阴影单元格为ghost层，标记“R”（“S”）的单元格属于中间接收（发送）缓冲区。后者在所有的方向上被重复使用。注意晕环总是提供给向上读取（不写入）的网格。为简洁起见，省略固定边界的单元

在 6 步晕环转换之后，当前网格 phi(:,:,:,t0) 的边界被更新，同时完成了在局部子区域上从 phi(:,:,:,t0) 到 phi(:,:,:,t1) 的 Jacobi 扫描（代码 37 行）。相应的子程序 Jacobi_sweep() 返回子区域上前一个时间步和当前时间步的最大偏差（一种 2D 情况下的可能实现请参看代码清单 6-5）。随后的 MPI_Allreduce()(代码 41 行) 计算了全局最大值并且使它全局可用，这样在所有进程上无需更多通信就可以决定是否由于到达收敛条件而停止循环迭代。 229

9.3.2 性能特征

MPI- 并行 Jacobi 解法器的性能特点对于很多区域分解代码都是典型的。我们区分对待强弱尺度场景，因为它们展示了相当不同的特征。所有的基准测试执行在两个不同的互联网络（Intel MPI 版本 3.2 上 DDR 高速互联与千兆位以太网）和一个基于 Intel Xeon 3070 处理器、主频 2.66GHz 的单槽节点的商用集群上。为了得到一个简单的尺度基准同时最小化内点影响，在每个节点上始终只用一个进程。

1. 弱扩展

在 5.3.6 节中我们的性能模型假定弱尺度的 3D 区域分解具有不变的通信开销。然而这也不总是正确的，因为位于网格边界的子区域可能需要通信的晕环会更少。幸运的是，由于子区域间内在的同步机制，并行程序的整体运行时间主要受限于最慢的进程，也就是在子区域等大小情况下拥有最大数量晕环的进程。因此，我们可以合理地预测线性扩展行为甚至是在很慢（非阻塞）的网络上，因为至少有一个子区域会在各个笛卡儿方向上都被其他子区域所环绕。图 9-10 展示的大小为 120^3 的常数子区域弱扩展数据证实了这个猜想。 230

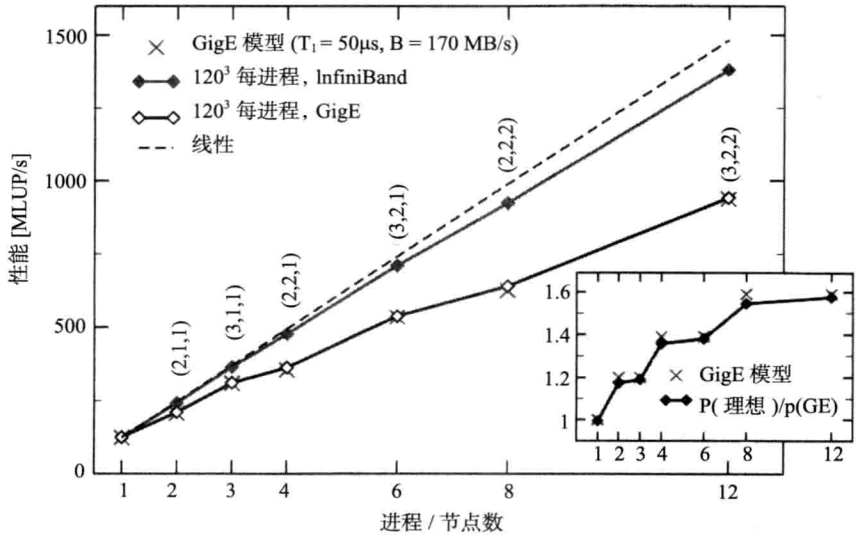


图 9-10 主图：问题规模为 120^3 每进程的 MPI- 并行三维 Jacobi 代码弱尺度对比——高速互联与千兆位以太网。每个节点运行一个进程。区域分解拓扑（每个笛卡儿方向上的进程数）也在图中进行了标注。弱尺度性能模型（“X”）能很好地重现 GigE 数据。插图：理想尺度性能和千兆位以太网相比于进程数目的比率（基于 Intel Xeon3070、主频 2.66GHz 的单套集群，Intel MPI 3.2 的单槽集群）

在高速互联网络上的可扩展性近乎完美。对于千兆位以太网，在节点数目很大的情况下通信仍然消耗了大约40%的整体运行时间，但当运行在少量节点上时这个影响则变得很小。事实上，性能图展示了一个奇怪的“缺口”结构，性能提升在进程数为4和8时有少量的下降。这些下降源于通信特征的基本改变，这个改变产生在任何坐标方向上的子区域的数量从一个变为多个的情况下。在那一点，开始设置沿着这个轴线的节点间通信：由于是周期边界条件，每个进程总是在所有的方向上进行通信，但是如果在一个特定的方向上只有一个进程，它将使用（快速）共享内存与自己交换晕环数据。图9-10中的插图指示了理想扩展与千兆位以太网性能数据之间的比率。很明显，比率随着新方向的切入而增大。这发生在分解方案（2,1,1）、（2,2,1）以及（2,2,2），分别属于节点数2、4、8。在这些点之间，比率都是常数。同时假定网络是非阻塞的，由于只有三个笛卡儿方向，因此即使在很多节点数目的情况下也可以预料到比率不会超过1.6。同样的现象也可以在极速互联数据上观测到，但是这个影响由于这种网络更大的带宽（×10）和更低的延迟（/20）而很少被提及。注意，尽管我们用了一个只与程序的并行部分有关的性能度量标准，但5.3.3节中提到的“假”弱扩展的考虑在此处并不适用；单CPU的性能与STREAM基准测试处于同等水平（参见3.3节）。

事实上对于一个定量的描述，上面提到的通信模型已经足够好了。我们假设点对点消息传递的基本性能特征可以通过沿着图4-10中线条的一个简单延迟/带宽模型描述。然而，由于在每个MPI进程上发送和接收晕环数据可以在6个笛卡儿方向上的每个方向重叠，所以我们必须包括一个最大的带宽数用于单链路上的全双工数据传输。（半双工）乒乓基准测试对于获得全双工带宽的合理估计是不够精确的，尽管大多数网络（包括以太网）声称支持全双工。千兆位以太网用在Jacobi基准测试上可以实现半双工通信111MB/s，全双工通信170MB/s，延迟为50 μs。

子区域的大小同样与进程数无关，因此对于在一次Jacobi扫描中所有单元更新的原始计算时间 T_s 也是常数。然而，通信时间 T_c 依赖于导致节点间的区域切分数量和大小，同时我们假定进程和它自身发生的复制到或者来自中间缓冲区的数据复制和通信没有任何开销。对于进程数为 $N=N_xN_yN_z$，总计 L^3N 个网格点的特殊问题大小（使用 L^3 个立方体子区域），其性能计算公式为

231

$$P(L,\vec{N})=\frac{L^3N}{T_s(L)+T_c(L,\vec{N})} \tag{9-1}$$

其中

$$T_c(L,\vec{N})=\frac{c(L,\vec{N})}{B}+KT_\ell \tag{9-2}$$

这里，$c(L,\vec{N})$ 表示在一个节点的网络链路上双向传输的最大数据容量，B 为全双工带宽，k 为（在所有子区域上）进程数多余一个的坐标方向的最大值。$c(L,\vec{N})$ 可以由笛卡儿分解导出：

$$c(L,\vec{N})=L^2\cdot k\cdot 2\cdot 8 \tag{9-3}$$

取 L=120，可以得到下面的数值：

N	(N_z, N_y, N_x)	k	$c(L, \vec{N})$ [MB]	$P(L, \vec{N})$ [MLUP/s]	$\frac{NP_1(L)}{p(L, \vec{N})}$
1	(1,1,1)	0	0.000	124	1.00
2	(2,1,1)	2	0.461	207	1.20
3	(3,1,1)	2	0.461	310	1.20
4	(2,2,1)	4	0.922	356	1.39
6	(3,2,1)	4	0.922	534	1.59
8	(2,2,2)	6	1.382	625	1.59
12	(3,2,2)	6	1.382	938	1.59

$P_1(L)$ 为在一个大小为 L^3 的区域上测得的单处理器性能。对于 $P(L, \vec{N})$ 的预测位于表格的第三列，表格的最后一列量化了相对于理想扩展的“减速因子”。这两列数据用于分别与图 9-10 中的主图和插图得出的测量数据进行比较。很明显，这个模型可以很好地描述弱扩展的性能特征，它表明我们通常的通信概念相比于计算“流程”是正确的。为了突出通信的重要性，这里我们故意选择了一个小规模的问题，但是延迟的影响也同样是次要的。

2. 强扩展

图 9-11 展示了在两个不同问题大小（120^3 与 480^3）上周期边值条件的强扩展性能数据。可以看出，独立于互联，即使在只有一个处理器的情况下小规模的问题也会有一些性能的缺失（大约 10%）。对于高速互联，两种大小不同的问题的性能间隙的产生主要由于不同的子区域大小。对于这种网络，考虑到节点数目，通信在可扩展性上的影响是次要的。然而对于千兆以太网，小问题的扩展性明显变差是由于晕环数据的体积（以及延迟开销）占所有工作量的比率随着节点增多而变大，导致在这个慢网络上通信掌控了性能。扩展曲线中的典型“锯齿”模式随进程数的变化与通信体积的改变是叠加的。这里使用和弱扩展一样的简单预测模型是不够的；尤其在小规模的网格上，强扩展对于子区域大小上的单进程性能具有很 232
强依赖性，同时内点通信（晕环数据交换）的进程变得很重要。更多的讨论参见习题 9.4 及 10.4.1 节。

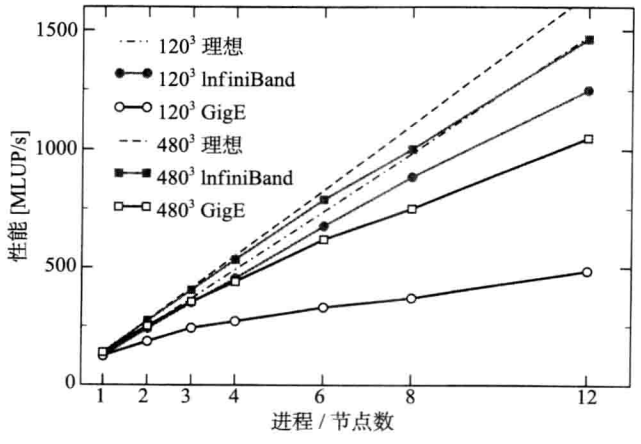


图 9-11 问题规模分别为 120^3（圆圈）和 480^3（方块）的 MPI 三维 Jacobi 代码在 IB（实心符号）网络和 GigE（空心符号）网络上的强扩展性对比。每个节点只发起一个进程（与图 9-10 中使用同样的系统和 MPI 拓扑结构）

注意按照当前并行系统的惯例，在处理单节点上多 MPI 进程的问题时类似的分析必须

精确（参考 4.4 节）。尤其在快速网络上，内点通信特征起主要作用（细节详见 10.5 节）。此外，在一个进程内从中间缓冲区复制数据和将数据复制到中间缓冲区亦不可忽视。

习题

9.1 *转换和死锁*。图 9-2 中展示了由环形转换引起死锁的补救方法（交换发送 / 接收顺序），请问这种解决方法对于进程数为奇数的情况是否也管用？如果链条是开的，即进程 0 不与拥有最大进程号的进程通信，会发生什么？在这种情况下，重排序发送和接收进程会有很大影响吗？

9.2 *死锁及非阻塞 MPI*。为了避免死锁，在 MPI- 并行 Jacobi 代码中我们使用非阻塞接收处理晕环数据交换（见 9.3.1 节）。实际上，MPI 的实现没有要求必须支持通信与计算的重叠；MPI 的进展，例如真实数据传输，很可能只在 MPI 库代码执行的时候发生。在这种条件下，还能保证死锁不
233 会发生吗？如有疑问，请参阅 MPI 标准。

9.3 *开边界条件*。9.3.2 节中的 Jacobi 代码的弱扩展性能模型假定问题具有周期边界条件。对于开边界条件（狄利克雷类型）模型会怎样变化？图 9-10 中的插图还会有一个停滞期吗？进程由 12 个增至 16 个时会发生什么变化？理想和真实性能比达到最大时的最小进程数是多少？

9.4 *并行 Jacobi 代码的强扩展性能模型*。正如 9.3.2 节中提到的那样，精确预测 MPI- 并行 Jacobi 代码强扩展行为的性能模型要比弱扩展模型复杂。尤其在子区域大小上对单进程性能的依赖性很难被预测，这是因为它取决于很多因素（流水线作用、预取、空间阻塞策略、到中间缓冲区的复制等）。这对于弱扩展模型不成问题，因为它拥有固定的子区域大小。然而，人们可以通过测量在所有子区域大小上并行运行的单进程性能来尝试建立一个部分“现象”模型，并以此为基准建立并行性能预测的标准。对于弱扩展模型，你认为还有哪些方面需要增强？考虑随着 N 的增长 T_s 逐渐变小，并且晕环交换不只发生在正在进行的节点间通信上。

9.5 *MPI 正确性*。下面的 MPI 程序片段正确吗？假定只有两个进程正在运行且 my_rank 包含了每个进程的进程号。

```
1 if(my_rank.eq.0) then
2    call MPI_Bcast(buf1, count, type, 0, comm, ierr)
3    call MPI_Send(buf2, count, type, 1, tag, comm, ierr)
4 else
5    call MPI_Recv(buf2, count, type, 0, tag, comm, status, ierr)
6    call MPI_Bcast(buf1, count, type, 0, comm, ierr)
234 7 endif
```

（例子来源于 MPI 2.2 标准文档 [P15]。）

高效 MPI 编程

很多 MPI 代码中隐藏着大量的优化潜能。在利用第 2、3 章提到的方法确保单进程性能接近最优之后，一个 MPI 程序应该通过基准测试来衡量它的性能及可扩展性以揭露与并行化相关的任何问题。它们中的一些与消息传递或者 MPI 自身无关，或产生自众所周知的一般性问题，如顺序执行（Amdahl 定律）、负载失衡、不必要的同步以及其他所有影响并行编程模型的作用。然而，也有一些特定问题与 MPI 息息相关，它们中的很多由隐式的关于分布式存储并行化的不合理假设或者对于通信的成本和副作用的过度乐观而引起。用户需要牢记一点，尽管 MPI 被设计为提供可移植同时高效的消息传递函数，但是给定代码在不同平台之间的性能是不可移植的。

本章尝试概述与高效 MPI 编程最相关的指南，这对于所有平台及 MPI 实现都具有不同程度的益处。由于每个算法的独特性，这样的一个概述必然是不完整的。和先前有关优化的章节一样，我们从对一个典型的分析工具的介绍开始，它能够检测出消息传递程序中的并行性能问题。

10.1 MPI 性能工具

与串行编程不同，通过简单地查找手册通常不可能检测出 MPI 性能问题的根源所在。

关于高级 MPI 分析的工具好在有一些免费的和商用的，可参见 [T24,T25,T26,T27,T28]。作为第一步，人们通常尝试获得与应用程序代码有关的 MPI 库运行时间、调用哪些函数，以及所需通信量的大致情况。这些数据至少可以揭示出通信是否成为瓶颈。IPM[T24] 就是一个能够获取这些信息的简单又低开销的工具。类似于大多数的 MPI 分析工具，IPM 使用 MPI 分析接口，这些接口属于标准的一部分 [P15]。每个 MPI 函数都是对实际函数的一个细微的封装，它们以“ PMPI_”开头。因此，一个预装的库或者甚至用户代码就可以中断 MPI 调用，同时收集分析数据。对于 IPM，预装一个动态库（或者链接一个静态库），然后运行应用就足够了。数据量（每个进程及每个进程对）、MPI 调用的时间开销、负载不均等信息在应用运行时累计，最后以图表的形式展现出来。图 10-1 展示了一个类似于 IPM 生成的主从工作类型应用的“通信”平衡图。每个柱状条对应于一个 MPI 进程号（rank），表示了进程在不同 MPI 函数上的运行时间。把这些时间与程序的整体运行时间进行对比是很重要的，因为如果程序的运行时间长达几个小时，那么 20 秒的同步时间是微不足道的。本例中，运行时间为 38 秒。进程 0（主进程）在各个工作者间分配任务，因此它在调用 MPI_Recv() 上耗费了大量的运行时间来等待结果。工作者们负载相当不平衡，它们运行时间中的 5% ～ 50% 被浪费在了栅栏处等待。参数上一点小的改动（减小工作包大小）可以改正这个问题，得出的平衡图如图 10-2 中所示。整体运行时间减少了大约 25%。

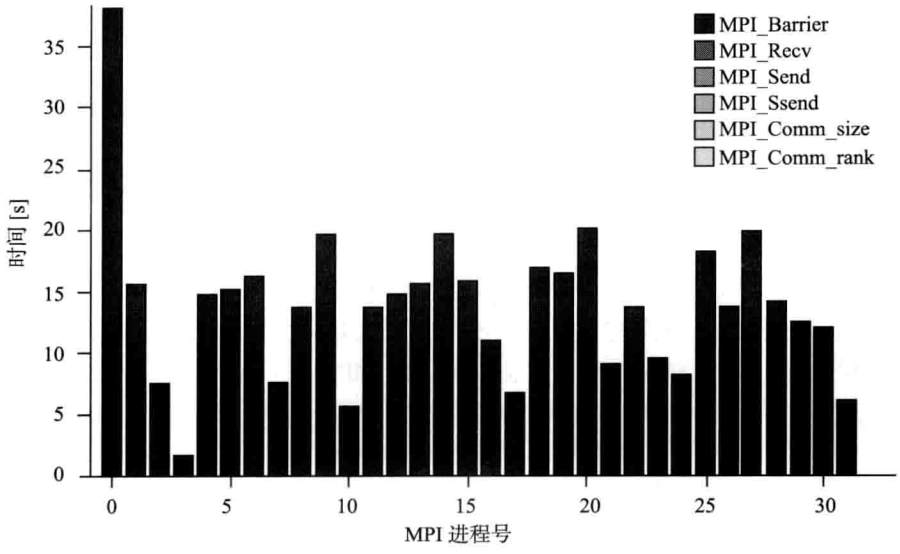


图 10-1 IPM 对于一个主从工作类型的并行应用“通信平衡”分析。全部的运行时间大约为 38 秒，几乎全部花在进程 0 的调用 MPI_Recv() 中。其他进程的负载十分不平衡，在一个栅栏内仅占到了 10% ～ 50% 的运行时间

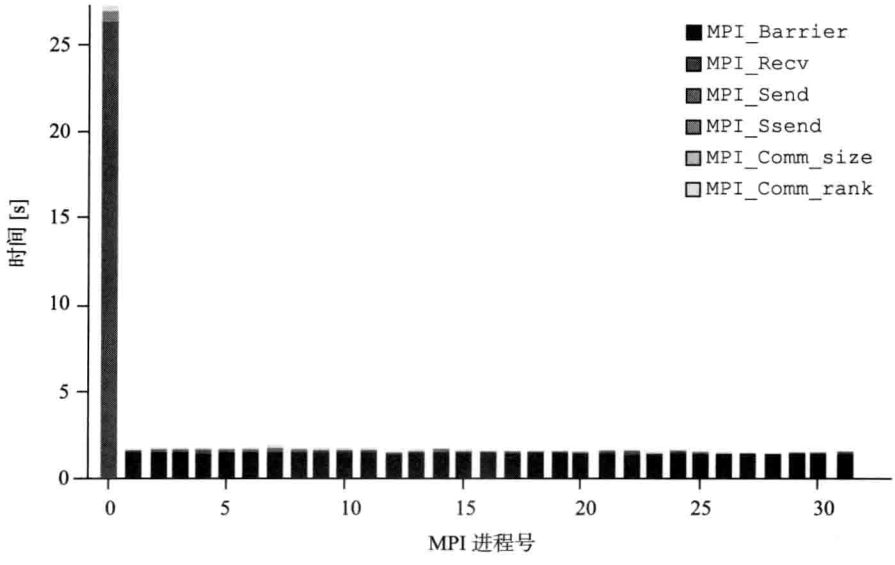


图 10-2 与图 10-1 相同应用的 IPM 函数分析，消除了其中的负载不均衡问题

注意，当解释和总结占用一个应用的全部运行时间结果时必须小心。本质上，同样的预定应用在了全局硬件性能计数器信息上（参见 2.1.2 节）。IPM 有一个小的 API，用于收集被分解成用户可定义阶段的信息，但是有时我们需要更详细的信息。更先进的工具能够支持一个成为事件时间表的功能。一个 MPI 程序可以被分解成非常专门的事件（消息的发送 / 接收、集合操作、阻塞等待等），它们能够很容易地用时间轴的方式可视化展示出来。图 10-3 是来源于“ Intel Trace Analyzer”的一个截图 [T26]。Intel Trace Analyzer 是一款 GUI 应用软件，它允许对 MPI 程序写入的轨迹数据进行浏览和分析（代码在运行前必须链接到一个专门的收集器库）。最顶层面板展示了一个类似于 9.3.1 节中 MPI 并行 Jacobi 解法器代码时间轴的

236

缩放图。图中的黑线表示点对点通信，亮线代表集合通信。沿时间轴，每个进程被分解成 MPI 代码（亮线）和用户代码（黑线）两部分。本例中运行时明显由 MPI 通信占主导。左下方的饼状图总结了在每个进程中用户代码和 MPI 代码各自占用的时间比例，作为负载不均衡的一种证据（本例中代码的负载是均衡的）。最后在右下方展示的面板里，可以读出每种组合下成对进程间的数据交换量。所有的这些数据均可以更详细地展示出来。例如，每个消息的所有相关参数和属性如持续时间、数据量、源和目标等都可以各自观察到。图中有 MPI 参与的部分可以被分开单独展示 MPI 函数，用户代码可以被仪表化，这样不同的函数便在时间轴和总结图中展示出来。

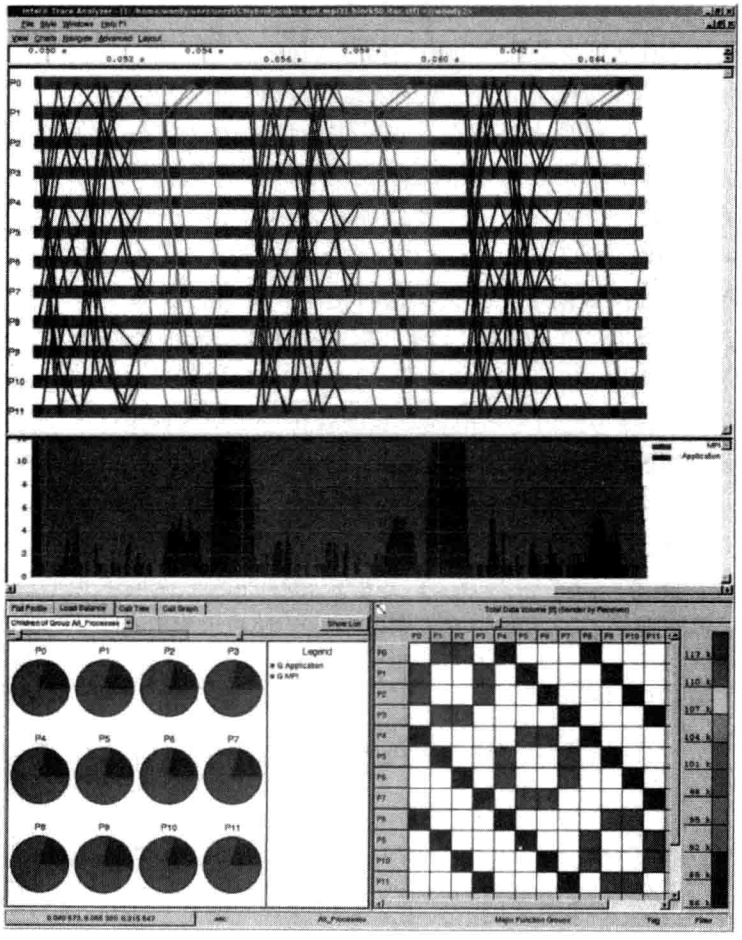


图 10-3 对一个运行在 12 个节点上的 MPI－并行代码使用 Intel Trace Analyzer 得到的时间轴图（最上方）、负载性能分析（最下方）以及通信总结（右下方）。在时间轴图中点对点（集合）消息用黑（亮）线条表示，黑（亮）框或扇形图代表执行的应用（MPI）代码

237
~
238

Intel Trace Analyzer 仅是众多商业和免费的 MPI 分析工具中的一个。尽管不同的工具可能会侧重于不同的方面，但是它们有着相同的意图，就是使能够代表 MPI 代码性能属性的大量数据变得更容易理解。一些工具侧重于大规模的系统，这时查看单个进程的时间轴则毫无意义；它们试图提供一个高层次的概况，同时数据自动给出一些优化建议。这是目前仍活跃研究的一个领域 [T29]。

高效使用MPI分析工具需要大量的经验，不了解消息传递代码基本性能陷阱的初学者是无法很好地使用它们。因此，本章的后半部分将会重点阐述这些内容。

10.2 通信参数

在4.5.1节中我们介绍了一些网络的基本性能属性，尤其侧重于点对点消息传递。尽管简单的延迟/带宽模型（见4.2节）相当好地描绘了有效带宽的总量特征，然而只有参数化地符合乒乓基准测试数据并不能重现正确（测出）的延迟值（见图4-10），原因在于MPI的消息传递要比我们过于简单的模型所涵盖的内容更加复杂。大多数的MPI实现版本在不同变量间的转换依赖于消息的大小以及其他因素：

- 对于短消息，消息自身及任何的附加信息（长度、发送者、标签等，也称作信封）可以被发送和存储在接收方预先分配好的缓冲区内而无需接收方干预。相匹配的接收操作不是必须的，但消息必须在一点从中间缓冲区复制到接收缓冲区。这也称为急切协议（eager protocol）。使用它的优势是可以避免同步开销。另一方面，它也需要大量的预分配缓冲区空间。一个使用很多急切消息的进程会因此溢出缓冲区，从而导致竞争或者程序垮掉。
- 对于长消息，缓存数据毫无意义。这时接收方将直接存储信封，但是真正的消息传递直到接收方缓冲区可用时才开始。额外的数据复制因此而避免，从而增加了可用带宽，但是发送方和接收方必须同步。这称作集结协议（rendevous protocol）。

取决于具体应用，很可能需要适应消息长度处在急切协议和集结协议适合的过度范围情况，或者需要为了急切数据而增加预留的缓冲区空间（在大多数MPI实现版本中这些都是可调的参数）。

函数MPI_Issend()可以使用在出现“急切溢出”这种问题的情形下。它工作起来和MPI_Isend()类似，除了一点不同的语义：如果发送缓冲区根据请求句柄可以再次使用，那
239 么此时发送－接收的握手过程已经完成，同时消息传递也已经开始，详见习题10.3。

10.3 同步、串行化和竞争

本节阐述一些不限定在消息传递上但可能以MPI的特殊形式出现的性能问题，因此值得详细讨论。

10.3.1 隐式串行化和同步

“非故意”的频繁同步抑甚至是串行化是并行编程中的常见现象，并不只限于MPI。在7.2.3节中我们已经演示了错误使用OpenMP同步并行代码是如何串行执行的。类似的陷阱也存在于MPI中，它们通常是由于错误地假设了消息的传递方式而产生。

在9.2.2节中用于阐明使用阻塞同步点对点消息造成死锁危险的环形通信模式是一个很
240 好的例子。如果链条是开的，那么环形将变成线性模式，此时消息在进程间的发送和接收以图10-4所示的顺序执行，这将不会产生死锁：进程5开始接收，与其匹配的发送位于进程4。当消息发送结束后，进程4可以开始响应它的接收操作。假定参数设置函数MPI_Send()为非同步的，并且可以使用“急切交付”（见10.2节），那么类似于MPI性能工具展示的那样，一个典型的时间轴图如图10-5所示。若网络为非阻塞的，则消息传递可以相互

重叠，同时由于所有的发送操作很早就结束（同阻塞发送一样快），所以大部分时间都花费在接收数据上（注意这里并没有指明数据精确存在的位置——它们可以在发送方到接收方之间的任意位置，这取决于 MPI 的具体实现）。

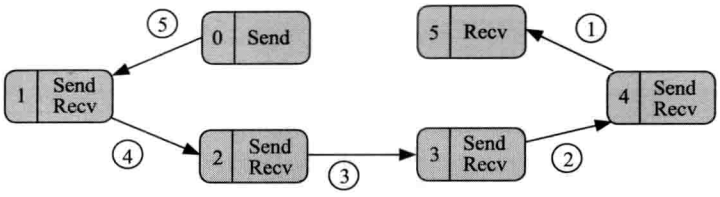


图 10-4 一种线性转换通信模式。即使是同步的点对点通信，死锁也不会发生，但是所有的消息传递将按图中所示的顺序串行执行

然而，这种模式存在一个严重的性能问题。如果消息的参数（最主要是长度）使得 MPI_Send() 实际的执行效果同 MPI_Ssend() 一样，则特有的同步发送语义一定会被观察到：在匹配的接收被响应前，MPI_Ssend() 不会返回用户代码。这不意味着 MPI_Ssend() 会一直阻塞直到消息被完整地传输并且到达接收缓冲区。因此，发送操作及与其相匹配的接收仅会有一小部分的重叠，虽然这能提供一些网络的并行使用，但同时也导致了一部分性能损失（时间轴图参见图 10-6）。这种情况的必要前提是消息传递仍然遵循急切协议：如果急切交付的条件被履行，则在接收操作响应前数据已经离开了发送缓冲区（以阻塞语义的形式），因此即使是在另一方终止接收的情况下同步发送也是安全的。

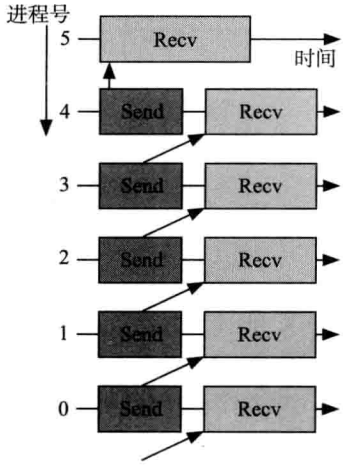


图 10-5 使用急切交付的阻塞（非同步）发送和接收线性转换时间轴视角（参见图 10-4）。利用非阻塞网络消息传递能够重叠。急切交付允许在对应的接收被处理前结束发送过程

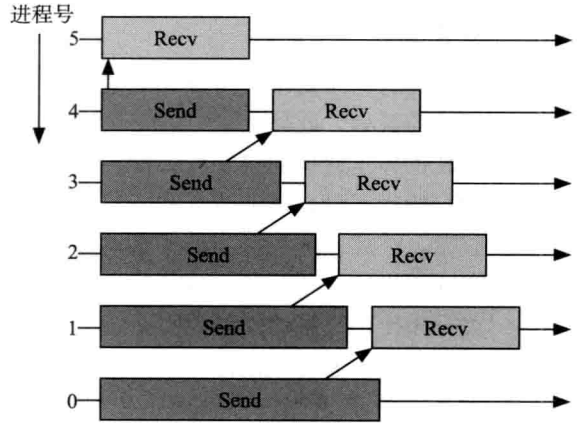


图 10-6 使用急切交付的同步阻塞发送和接收线性转换时间轴视角（参见图 10-4）。虽然消息传递（箭头）可以被完美重叠，但消息的发送却只在与其匹配的接收操作响应后结束

当消息遵循集结点协议传输时，情况将变得更糟。在这里用缓冲区是不可行的，因此发送方和接收方必须以一种方式同步以确保完整的端到端数据交付。在我们的例子中，五个消息将会一个接一个地串行传输，这是因为没有进程可以在下一个进程完成接收前完成消息的发送。进程位于的地方沿链条越远，自身的同步发送操作阻塞的时间就会越久，并且没有重叠的可能性。图 10-7 用时间轴图说明了这种情形。

241

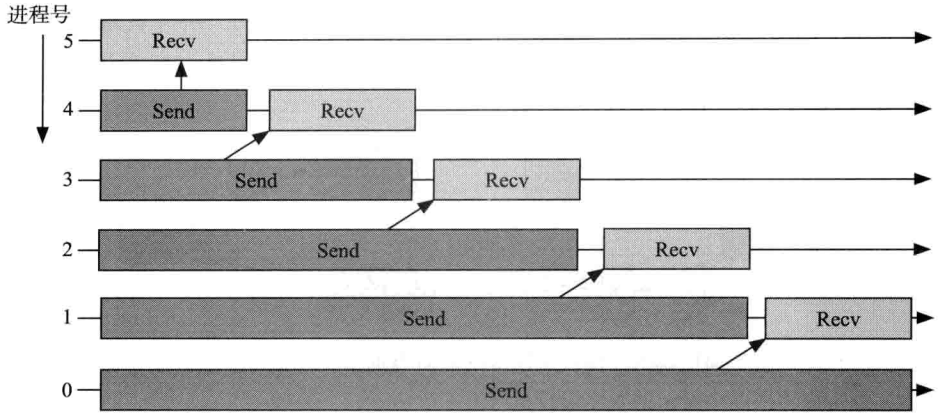


图 10-7 使用集结点协议的阻塞发送和接收的线性转换时间轴视图（见图 10-4）。由于无法使用缓冲区，消息（箭头）串行传递

隐式串行化应该尽量避免，因为它不但是一种额外的通信开销，还有可能像上面那样导致负载不均衡。因此，思考清楚如何转换（环形或线性）是很重要的，这种转换的同时相似的模式可以被高效执行。一些基本的替代方案在 9.2.2 节中已经进行了描述：

- 改变进程上的发送和接收操作次序，例如，所有的奇数号进程（见图 10-8）。成对的进程因此能够利用可用网络容量的一部分并行交换消息。
- 使用如 9.3 节在并行 Jacobi 解法器中展示的非阻塞函数。由于多个通信可以被 MPI 库以一种最优的次序处理，所以非阻塞函数可以获得额外的益处。此外，它们还提供了至少一种实现真正异步通信的机会，即使当一个进程在执行它的代码时，辅助线程和硬件机制可以传输数据。注意，这种操作模式必须被看作是 MPI 实现提供的优化手段；MPI 标准故意避免了有关异步传输数据的任何说明。
- 使用不会引起死锁的点对点函数，忽略消息的大小，特别是 MPI_Sendrecv()（见习题 10.7）或 MPI_Sendrecv_replace()。在内部，这些函数调用通常是由非阻塞调用和 MPI_Wait() 的组合来实现，实际上这些属于方便使用的函数。

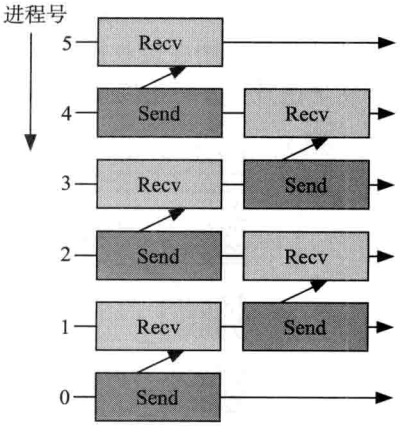


图 10-8 即使消息发送是同步的并且使用集结点通信，在奇数号进程上交换发送和接收次序也会展露出通信的并行性

242

10.3.2 竞争

我们已经用过的简单延迟 / 带宽通信模型加上细致的消息传递（见 10.2 节）能够解释很多作用，但并不包括竞争作用。这里我们想将有关竞争的讨论约束在网络互联上；忽略共享内存多槽多核系统上的共享资源，例如一个计算节点（有关这些问题的讨论参见 1.4 节、4.2 节及 6.2 节中的例子）。假定有一个设计成 4.4 节中讨论的那种典型混合（分层次）并行计算机，网络竞争将发生在两个级别上：

- 同一个节点上的多个线程或进程会向其他节点发出通信请求。如果带宽没有扩展到多互联，则每个连接上的可用带宽将会下降。这对于通常只有一个 MPI 通信的（有时甚至需要访问远程文件系统）网络接口的商用系统是非常普遍的。在这些机器上，单一的线程就能够占据网络接口。然而，当需要多互联来充分利用可用的网络带宽时，也会使用并行计算机 [O69]。
- 网络拓扑可能不是完全非阻塞的，即分开的带宽（见 4.5.1 节）可能会低于节点数与单个连接的带宽之积。这在如立方网格或肥树非阻塞网络中是很常见的。
- 即使分开的带宽是最优的，但对于某些特定的通信模式（见 4.5.3 节中图 4-17）静态也会导致竞争。对于已经定义好通信框架且需要性能优化的单独应用，改变网络搭建好的路由表（如果可行）可以成为一个选择 [O57]。

除了最罕见的消息传递方式，通常情况下，在当前的并行系统上很难避免一些类型的竞争。只有在通信代表了运行时的一个可测量部分时，对应用性能的实际影响才会显现。 243

注意有些通信模式特别容易引起竞争，如每个进程向其他所有进程发送消息的多对多通信传输模式；MPI 中的函数 MPI_Alltoall() 便是这种情况的一种特殊形式。可以预见，在未来的几年里，大规模并行体系结构下的多对多通信性能会继续下降。

任何通过降低通信开销及消息传输量（见 10.4 节）的优化手段都很有可能对减少竞争有效。即使无法减少消息数据的传输量，也可以通过重排通信调用的次序来使竞争最小化 [A85]。

10.4 降低通信开销

10.4.1 最优化区域分解

区域分解是数据并行化最重要的实现方式之一。大多数流体动力学及结构力学的模拟均基于区域分解及晕环层交换。9.3.2 节中我们已经证明了在简单情况下，晕环通信的性能特点可以被模型精确地模拟，同时将整个问题在子区域上进行的分解决定了通信的数据量，这将严重影响性能。下面我们将要弄清楚选择一个“错”的分解（开销方面）会造成哪些影响，同时阐明如何导出 9.3.2 节描述的并行 Jacobi 解法器中的性能模型。

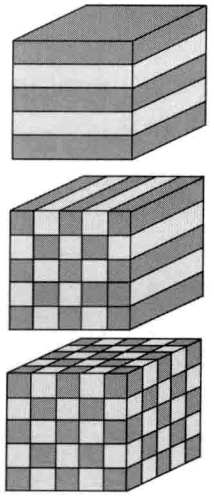


图 10-9 周期边界条件、问题大小为 L^3（强可扩展）的立方区域三维区域分解：切分成一维（最上面）、二维（中间）或者全部三个维度（下面）三种情况下，每个进程的通信量 $c(L, N)$ 与进程数 N、单地域数据量 w（字节）之间的关系

1. 最小化区域间表面积

图 10-9 展示了 L^3 大小的立方体区域在强扩展下分解成 N 个子区域的不同的可能情况。取决于区域切分执行在一维、二维或者所有三个维度（从顶到底）的不同情况，每个进程同它相邻进程产生的必要通信数据量 $c(L, N)$ 也随 N 不同有相应的变

化。最好的情况是在立方子区域上获得的（参见习题 10.4），如最速下降法。此处我们忽略分解选项对 N 和整体区域形状（可能不是立方体）的依赖。在假设完整区域为立方体的前提下，函数 MPI_Dims_create() 默认会尝试使子区域尽可能地变成方块。尽管“平板”子域更易实现，但我们通常情况下都不会使用它，主要原因在于相比于网柱或立方体区域分解，它产生与 N 无关的更大开销。每个子域上的通信量为常数会严重损害强可扩展性，因为性能
244 在较低级别上达到饱和，这受限于消息大小（如平板面积）而非延迟（见式（5-27））。

在网柱形和立方体形子域通信量表达式中，N 的负指数幂会抑制通信开销，但数据表面积与体积的比率仍会随着 N 增大而增大。更糟糕的是，对于常量问题规模的处理器数目扩展会使得“乒乓曲线”沿着更小的消息量方向下降，最后变成延迟主导的方式（见 4.5.1 节）。这些已隐含地在我们的精确性能模型（见 5.3.6 节，特别是公式 5-28）和“慢计算”（5.3.8 节）中予以考虑。注意，在没有消息重叠的作用下，六个晕环通信中的每个均遭受了延迟；如果延迟占据开销的主导地位，那么“最优”三维分解将会由于每个区域有更多的邻居进程而变得不尽如人意。

每个位置（w）的通信量取决于具体问题。对于 9.3 节中的简单 Jacobi 算法，w=16（使用双精度浮点数，沿正负坐标方向各 8 字节）。如果算法需要高阶导数或者存在长距离的相互作用，那么 w 将会更大。同样的情形适用于当网格点的数据结构比标量更复杂的情况，
245 例如 lattice-Boltzmann 算法 [A86,A87]。也可参见下面的章节。

2. 映射问题

当代计算机毫无疑问全都具有分层次的机构。它们全都是由共享内存的多处理器“节点”通过网络耦合而成（参见 4.4 节）。使用这种硬件的最简单方法就是在每个核上运行一个 MPI 进程。假设位于相同节点上的任意两核间的点对点通信均大大快于跨节点间的两核通信（涉及带宽和延迟），显然计算子区域到核的映射对通信开销具有重要影响。理想情况下，如果允许进程重排，那么映射会被函数 MPI_Cart_create() 优化，但是大多数 MPI 实现并不知道并行计算机的拓扑结构。

举个简单的例子来说明这一点。物理问题为在一个具有周期边界条件的 4 × 8 笛卡儿进程网格上进行二维仿真。图 10-10 描述了一个在 4 个节点（A ～ D）、每个节点 8 个核的机器上的典型“默认”配置（我们略去了网络拓扑以及任何如 cache 组、ccNUMA 局部域等这样的子结构）。基于内点互联开销低的假设，相邻点的通信效率（如晕环层交换）取决于每

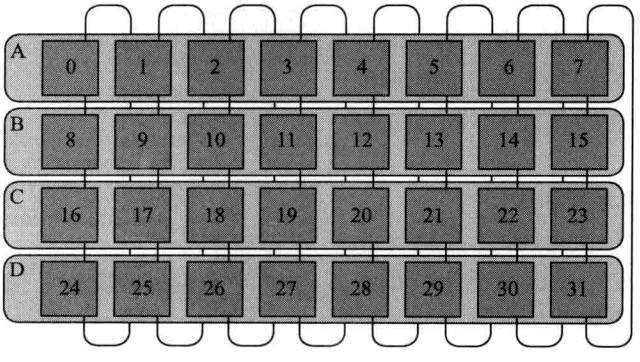


图 10-10 一个二维 4 × 8 周期区域分解的典型 MPI 进程号（数字）到子区域（方块）和集群节点（字母）上的默认映射。每个节与其他节点有 16 个接口相连。内点间的连接被省去

个节点上节点间相连的最大数目。图 10-10 上的映射导致了 16 个这样的连接。每个节点上的“通信表面”要比它需要的大，这是因为分给它的 8 个子域沿着一个维度线性排开。选取图 10-11 中的短长方形策略可以立即将节点间的互联数降到 12 个，从而减少网络竞争。由于每个连接的数据量保持不变，这就相当于减少了 25% 的节点间通信量。实际上对于这个问题，这已经是能找到的最小开销的映射方案。

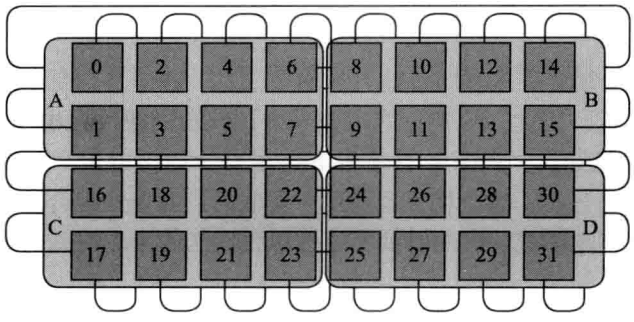


图 10-11 MPI 进程号和子域到节点上的一种完美映射。每个节点同其他节点有 12 个节点间互联

到目前为止，我们一直预先假设 MPI 子系统会尽可能优先在同一个节点上分配连续进程号，例如，在使用下个节点前填满当前节点。尽管这在很多并行计算机上都是一个合理的假设，但这绝不能认为是理所当然的。考虑连续进程号映射到连续节点这样的循环分布情况，此时图 10-11 中的最优解将会变为最差的方案：图 10-12 表明此时每个节点含有 32 个节点间互联。 246

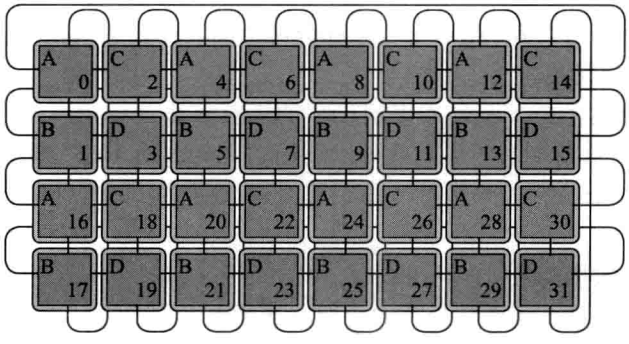


图 10-12 与图 10-11 中相同的进程号到子域间的映射，但进程号以循环的方式被分配到节点上，即连续的进程号运行在不同的节点上。这样导致了每个节点上具有 32 个节点间互联

类似的想法应用在其他类型的并行计算机上，例如基于（超）立方体网格网络的结构（见 4.5.4 节），这种结构上的点间通信通常是有利的。如果笛卡儿拓扑结构与 MPI 进程号到节点间的映射不匹配，那么产生的长距离连接会导致难以忍受的缓慢通信（通过网络容量衡量）。当然，对于应用性能的实际影响可能会变化。如果并行程序不被通信所限制，那么任何类型的映射都是可接受的。然而，最好牢记一件事，就是 MPI 环境提供的默认设置通常不可信。可以使用 10.1 节中介绍的 MPI 性能工具列出每个单独的点对点连接的有效带宽，这样如果数字与期望不符，就可以诊断出可能的映射问题。

截至目前，我们一直忽略任何内点连接问题，并假设在同一节点上不同核之间的MPI通信“无限快”。尽管内点延迟的确要远小于目前存在的任何一种网络连接技术，但是带宽是一种完全不同的问题，并且不同的MPI实现在内点性能上大不相同，更多信息参见10.5节。真实的混合程序，即每个MPI进程由多个OpenMP线程组成的程序，映射问题变得更
247 加复杂。详细的讨论参见11.3节。

10.4.2 聚合消息

如果一个并行算法需要在进程间传输大量的短消息，通信将会变成受延迟制约的，因为每个消息都会导致延迟。因此，短消息应该被聚合到连续的缓冲区中然后以更大的块发送，这样将只需承受一次延迟，同时有效通信带宽会尽可能地接近于乒乓图上的饱和区域（见4.5.1节中的图4-10）。当然，这个优势也适用于类似的点对点及集合通信。

只有在复制消息到连续缓冲区的时间不超过单独发送的延迟时间的这种条件下，聚合才会奏效。即如果

$$(m-1)\,T_\ell > \frac{mL}{B_c} \tag{10-1}$$

这里 m 为消息的数量，L 为消息的长度，B_c 为内存到内存的复制带宽。为了简便起见，我们假设所有的消息具有相同的长度，并且内存间复制的延迟极其微小。实际获得的优势也依赖于原始网络带宽 B_n，因为串行通信与聚合通信的时间比率为

$$\frac{T_s}{T_a} = \frac{T_\ell/L + B_n^{-1}}{T_\ell/mL + B_c^{-1} + B_n^{-1}} \tag{10-2}$$

例如在一个慢网络上，如果 B_n^{-1} 相对于分子和分母上的其他表达式很大，这个比率就会接近于1，聚合通信将不会获益。

消息聚合的一个典型应用是使用多层晕环的模板解法器：在一个子域上经过多次更新（扫描）之后，一个消息的多晕环层交换可以利用“PingPong ride”来降低延迟的影响。如果这个方法对优化一个已有代码是可行的，那么应该使用合适的性能模型来估计期望增益[O53,A88]。

消息聚合和派生数据类型

典型的消息聚合的例子出现在分开的，即非连续数据项必须在进程间传输的时候，如矩阵的一行（Fortran）或者一串不相关的、很可能类型也不同的变量。MPI提供支持这种功能的所谓的导出数据类型（derived datatype）。程序员可以引入新的数据类型而不只局限于内建的种类（MPI_INTEGER等）并且在通信调用中使用它们。有大量的选择可供定义新的类型：带间距的类数组、索引数组、n 维数组的子数组，甚至还可以是分散在内存中毫无关联、不同类型的数据集合。新类型必须首先使用MPI_Type_XXXXX()定义，此处“XXXXX”指代上面提到的一个变量。函数调用将新类型作为一个整数类型（Fortran中）或者是一
248 个MPI_Datatype结构（C/C++中）返回。为了使用这个类型，必须先由函数MPI_Type_commit()提交。一旦这个类型不再需要了，可以使用MPI_Type_free()将其释放。

下面我们定义一个新类型用来表示Fortran矩阵中的一行以演示上面提到的内容。由于Fortran采用列主元来实现多维数组（见3.2节），因此矩阵行在内存中是不连续的，行中的每个元素被一个固定跨度的数据块隔开。这种情况下适合的MPI调用是MPI_Type_

vector()，它的主要参数在图 10-13 中描述。我们用它来定义 XMAXXYMAX 维的双精度矩阵中的一行。因此，count=YMAX，blocklength=1，stride=XMAX，原始类型为 MPI_DOUBLE_PRECISION：

```
1  double precision, dimension(XMAX,YMAX) :: matrix
2  integer newtype                                  ! new type
3
4  call MPI_Type_vector(YMAX,                       ! count
5                       1,                          ! blocklength
6                       XMAX,                       ! stride
7                       MPI_DOUBLE_PRECISION,       ! oldtype
8                       newtype,                    ! new type
9                       ierr)
10 call MPI_Type_commit(newtype, ierr)              ! make usable
11 ...
12 call MPI_Send(matrix(5,1),                       ! send 5th row
13               1,                                 ! sendcount=1
14               newtype,...)                       ! use like any type
15 ...
16 call MPI_Type_free(newtype,ierr)                 ! release type
```

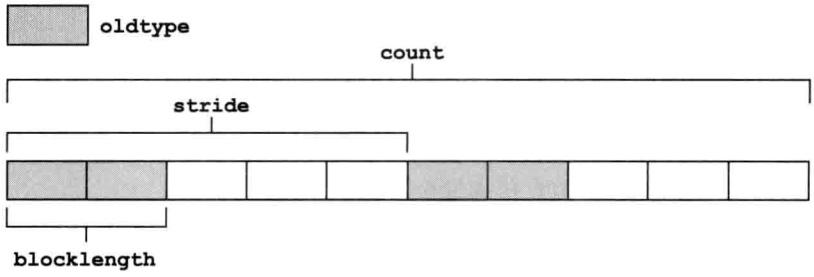


图 10-13　MPI_Type_vector 需要的参数。这里 blocklength=2，stride=5，count=2

代码 12 行中，这个类型被用于发送矩阵的第 5 行，MPI_Send() 中的 count 值设置为 1（当发送多于一个这种类型的实例时必须要小心，因为每个实例结尾处的“跨度”在默认情况下是被忽略的；细节请查阅 MPI 标准）。在笛卡儿拓扑结构中简化晕环交换的数据类型可以用相似的方式建立。

尽管派生数据类型方便使用，但是它们的性能隐患仍不清楚，这是性能优化在各版本 MPI 实现间不可移植这个原则的一个很好的实例。库可以将新类型的一部分聚合进一个内部 249
的连续缓冲区，但也可以分别发送这些片段。即使发生了聚合，人们也不能确保它以最高效的方式完成；例如使用非临时的存储区对较大量的数据有益，又如（如果每个 MPI 进程可以使用多个线程）复制可以是多线程的。一般情况下，如果导出数据类型对性能影响很大，用户则不应该依赖于库的效率，而是应该检验手动复制是否可以提高性能。如果可以，这个“性能缺陷”需要反映给 MPI 库的提供者。

10.4.3　非阻塞与异步通信

除了前面章节中提到的努力降低通信开销外，进一步增加并行程序效率的方法是将通信和计算的时间重叠。非阻塞点对点通信似乎是实现这一目标的直接手段，并且实际上我们已经在 MPI- 并行 Jacobi 解法器中使用了它，当时我们使用 MPI_Irecv() 函数来重叠复制数据到发送缓冲区的晕环数据接收和发送过程（见 9.3 节）。然而，在模板更新（包含实际的“工作”过程）和通信之间并没有并行性。一种实现这个目标的方法是首先执行形成子域边界的

模板更新，因为它们必须要被传输到相邻子域的晕环层。完成更新和将数据复制到中间缓冲区后，MPI_Isend() 可以被用于在模板更新执行的过程中发送数据。

然而，正如更早前提到的那样，用户必须严格地将非阻塞与真正的异步通信区分开来。根据 MPI 标准，非阻塞语义仅意味着消息缓冲区在函数调用从 MPI 库返回后不能继续使用；虽然是理所当然的，但是否数据传输完全取决于具体实现，即 MPI 过程发生在用户代码执行在 MPI 外面时。

代码清单 10-1 展示了一段可以用于检验一个 MPI 库是否支持异步通信的基准测试程序。代码恰好由两个处理器执行（我们略去了初始化代码）。do_work() 函数以参数指定的数秒钟的持续时间执行一些用户代码。为了排除竞争效应，函数应该执行一些不涉及同步内存传输的操作，如寄存器间的算术运算。MPI 的数据大小（count）选取为使消息传递耗费大量时间（数十毫秒）的尺寸，即便是在最先进的网络上。如果 MPI_Irecv() 发起了一个真正的异步数据传输，那么测量的整体时间会随着增大的延迟保持不变直到延迟等于消息的传输时间。除此之外，执行时间都会线性增加。另一方面，如果 MPI 过程只在 MPI 库内部发生（本例中意味着在 MPI_Wait() 内），数据传输的时间及 do_work() 的执行时间会一直累加，同时整体的运行时间会从零延迟开始线性上升。图 10-14 展示了一些当代并行计算机结构和互
250 联情况下节点间的互连节点数据（空心符号）。在这些机器系统中，只有 Cray XT 大规模并行系统默认支持异步的节点间 MPI 操作（空心钻形）。IBM Blue Gene/P 系统默认使用轮询的方式处理消息，从而排除了异步传输（空心方形）。然而，基于中断的过程可以在这台机器上被激活 [V116]，从而使异步消息传递发生（实心方块）。

代码清单 10-1 MPI 执行异步点对点通信能力的测试代码

```
double precision :: delay
integer :: count, req
count = 80000000
delay = 0.d0

do
  call MPI_Barrier(MPI_COMM_WORLD, ierr)
  if(rank.eq.0) then
    t = MPI_Wtime()
    call MPI_Irecv(buf, count, MPI_BYTE, 1, 0, &
                   MPI_COMM_WORLD, req, ierr)
    call do_work(delay)
    call MPI_Wait(req, status, ierr)
    t = MPI_Wtime() - t
  else
    call MPI_Send(buf, count, MPI_BYTE, 0, 0, &
                  MPI_COMM_WORLD, ierr)
  endif
  write(*,*) 'Overall: ',t,' Delay: ',delay
  delay = delay + 1.d-2
  if(delay.ge.2.d0) exit
enddo
```

人们也许会提出如果 do_work() 函数执行受边界约束的代码，那么结果将会改变，这是因为消息传输会影响 CPU 可用的内存带宽。然而，这种作用只有在网络带宽大到堪比一个节点的聚合内存带宽时才是主要的，但是这对于当今的计算机系统不成问题。

尽管在计算系统的选择上并没有对当代的技术进行彻底调查，但结果仍具有代表性。从商用集群到价格昂贵的超级计算机，几乎没有对异步非阻塞传输消息的任何支持，尽管大多数计算机系统都装备了像 DMA 引擎这样允许后台通信的硬件设施。这种情况对于内点消息

传递甚至更糟糕，因为用于内存间复制的专用硬件十分罕见。对于 Cray XT4，这种情形演示在图 10-14 中（实心钻形）。注意，纯通信时间大致匹配内点情形的时间，尽管此时不使用机器的网络并且 MPI 可以利用共享内存进行复制。这是由于长消息的 MPI 点对点带宽几乎等同于内点及节点间的情形，所以这个特征在混合并行系统上非常普遍。详见 10.5 节中的讨论。 251

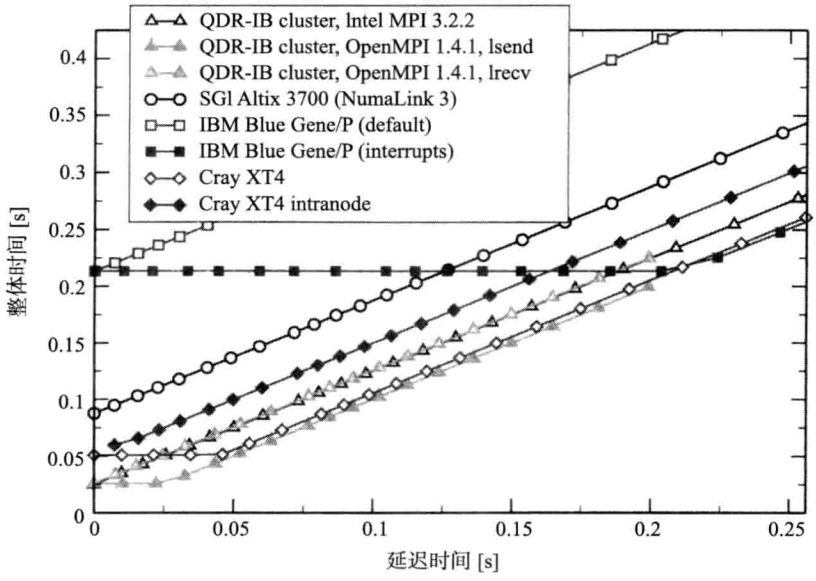


图 10-14 在不同体系结构中，连接方式及 MPI 版本下的 MPI 重叠基准测试结果。在所有这些结果中，只有 Cray XT4 上的 MPI 库（钻形）及高速互联集群上的 OpenMPI（实心三角形）能在默认情况下支持异步传输，然而 OpenMPI 对于非阻塞接收不允许重叠。内点通信中，重叠技术在这些考虑的系统中通常是不可用的。在 IBM Blue Gene/P 系统上，通过设置参数 DCMF_INTERRUPTS=1 启动中断驱动的过程（实心方形）可以实现异步传输

结论是人们不应该把过多的优化精力放在通过非阻塞点对点的函数调用而利用非阻塞通信上，因为这种手段只会在很少的环境中才会成功。然而这并不意味着非阻塞 MPI 毫无用处；它对于预防死锁、减少同步开销引起的闲置时间以及高效处理多个通信请求都是有价值的。稍后的一个例子便是当发送和接收操作同时出现时利用了全双工传输。与异步传输不同，全双工通信被当今的大多数连接方式及 MPI 实现所支持。

即使 MPI 不支持在其他线程执行用户代码时额外发起一个单独的线程（OpenMP 或者线程的其他变体）用于处理 MPI 调用，但将计算和通信进行重叠仍然是可能的。这是混合编程的一种变形，我们将在第 11 章中对它进行讨论。 252

10.4.4 集合通信

在 9.2.3 节中我们通过调用 MPI_Reduce() 替换“手动”将各部分结果累加的方式对数值积分程序进行了修改。除了普遍性地降低了编程的复杂性之外，集合通信也具有优化的潜质：程序最原始的组织方法使得通信开销随进程数的增加而线性增大，这是由于即便是使用非阻塞通信，在接收端仍然会有严重的竞争（见图 10-15）。如果网络是非阻塞，但将各部分

结果使用分组的进程相加然后再将结果传播给接收进程的这种“树状”通信模式能够改变对算法的线性依赖性（见图 10-16）。（这里我们将网络延迟和带宽以同样的方式对待。）尽管每个独立的进程通常不得不串行化它所有的发送和接收，但仍然会有足够的并行性使得树状模式比简单的线性方法更加高效。

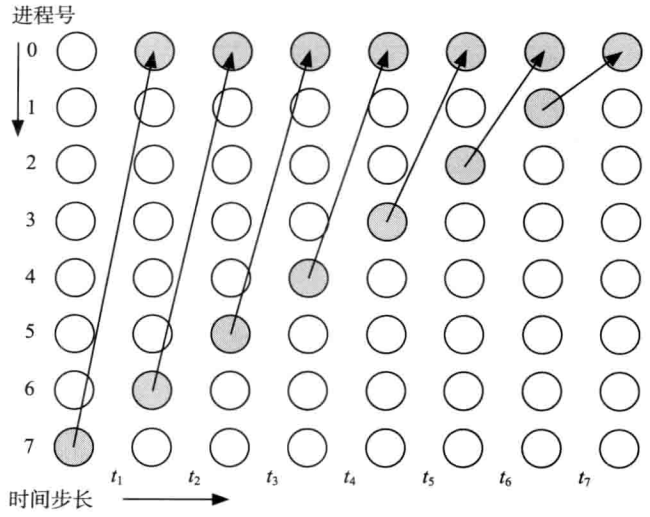


图 10-15 在积分例子（代码清单 9-3）中实现的通信开销与进程数量呈线性关系的全局归约。每个箭头代表了一个到接收者进程的消息。阴影部分表示在一个时间步长内通信的进程

集合 MPI 调用内置了恰当的算法以在任何网络上都能获得合理的性能 [137]。在理想情况下，甚至 MPI 库都能包含网络拓扑的足够信息以选取最优的通信模式。这就是在与简单的点对点通信实现相同功能的情况下优先选择集合通信的主要原因，参见习题 10.2。

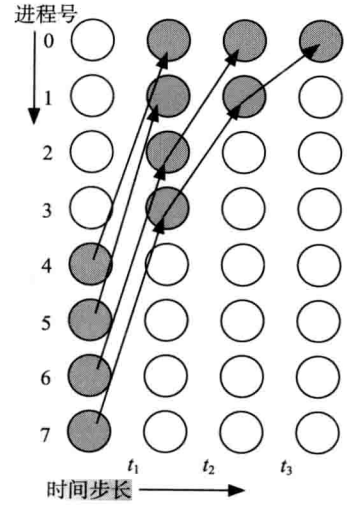


图 10-16 在树状分层归约模式下，通信开销与进程数成对数关系，这是由于通信在每个时间步上是并行的

10.5 理解节点内点对点通信

当在一个系统核间和节点间寻找最优的线程和进程分布时，通常假设任意的 MPI 内点通信都是无限快的（参见上面的 10.4.1 节）。令人意外的是，这种假设通常是不正确的，尤其是涉及带宽的时候。尽管今天一个单核就能使用掉芯片内存接口多个 GB/s 的带宽，但是 MPI 实现的低效使用内点通信仍会极大地损害性能。当 MPI 库没有意识到两个通信中的进程运行在同样的共享内存节点上时，在这方面最简单的“错误”就会出现。此时相对较慢的网络协议就会被使用以替换掉内存间的副本。即使 MPI 库在共享内存可用的情况下使用了共享内存通信，仍然有一系列可能的策略：

253

- 是否会使用非临时存储或 cache 行 0（见 1.3.1 节）很可能依赖于消息及缓存的大小。如果消息很小并且进程运行在一个 cache 组里，使用非临时存储通常事与愿违，因为

它会产生额外的内存运输。然而，如果没有共享的 cache 或者消息很大，那么数据将不得不写入主存中，这时非临时存储避免了将会产生的写分配。

- 数据传输可能是“单次复制”的，这意味着一次简单的块复制足以将消息从发送缓冲区复制到接收缓冲区中（隐式地实现了一个同步集合点协议），或者使用一个中间（内部的）缓冲区。稍后的策略需要额外的复制操作，它在网络很快的情况下可以大幅度减少通信带宽。
- 内点内存间的数据传输得到了硬件的支持。在共享 cache 对于通信性能不重要的情形下，使用专门的硬件设施会产生出色的点对点带宽 [138]。 254

上述问题的 MPI 库行为有时会受可调参数的影响，但是带有复杂分层的 cache 及系统设计的多核处理器结构的快速发展也使它成为了一个迅猛发展的主题。

同样，IMB 套件的简单乒乓基准测试（见 4.5.1 节）可以用来探寻内点 MPI 通信的性质 [O70，O71]。我们使用 Cray XT5 系统作为一个出色的例子。一个 XT5 节点包含两个 AMD 皓龙芯片，每个芯片上搭载了一个 2MB 的 4 核 L3 组。这些节点通过一个 3 维环形网络连接（见图 10-17）。由于这个结构，人们可以期待三个不同级别的点对点通信特征，这些特征依赖于消息传输是否发生在 L3 组内（内点插槽内）、不同插槽核间（内点插槽间）或者不同节点间（节点间）。（如果一个节点含有大于两个的 ccNUMA 局部域，将会有更多的变化。）图 10-18 展示了这个系统的节点间及内点的测试数据。同预计的一样，对于短的和中等长度的消息，节点间及内点的通信特征相当不同。同一插槽上的两个核能从共享的 L3 级 cache 上获得很大益处，导致了超过 3GB/s 的峰值带宽。意外的是，插槽间的通信特征十分相似（虚线），尽管没有共享缓存并且由于所有数据通过主存交换而不应出现高带宽“驼峰”。对这一特殊效应的解释存在于标准乒乓基准测试执行的方式中 [A89]。与 4.5.1 节中展示的伪 255

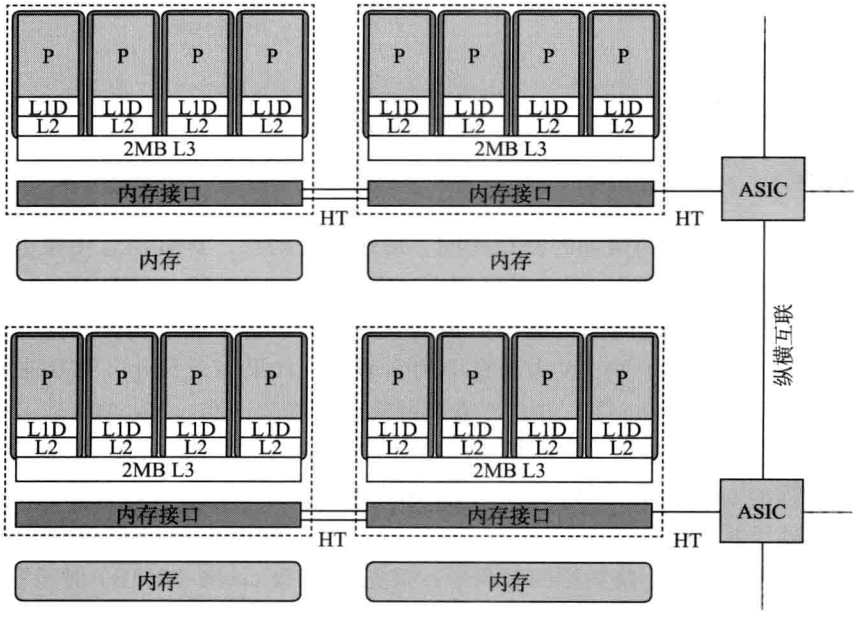


图 10-17 Cray XT5 系统的两个节点。虚线框代表 AMD 皓龙处理器插槽，每个节点上有两个插槽（NUMA 局部域）。纵横互联处实际上是一个三维环（网）网络

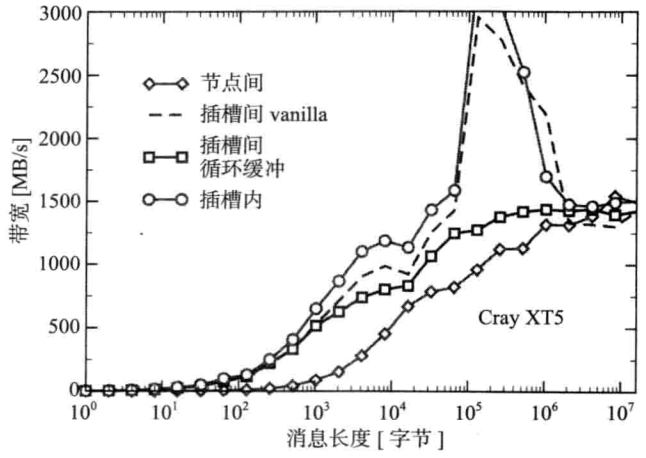


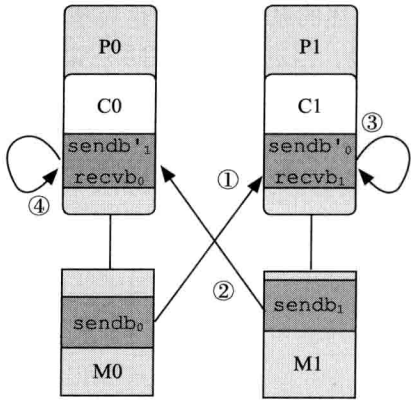
图 10-18 在 Cray XT5 系统上节点间、插槽间内点以及插槽内通信的 IMB 乒乓性能。插槽间“vanilla”数据的获得没有使用循环缓冲区（细节见正文）

代码不同，真正的 IMB 乒乓代码组织如下：

```
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
if(rank.eq.0) then
  targetID = 1
  S = MPI_Wtime()
  do i=1,ITER
    call MPI_Send(buffer,N,MPI_BYTE,targetID,...)
    call MPI_Recv(buffer,N,MPI_BYTE,targetID,...)
  enddo
  E = MPI_Wtime()
  BWIDTH = ITER*2*N/(E-S)/1.d6      ! MBytes/sec rate
  TIME   = (E-S)/2*1.d6/ITER        ! transfer time in microsecs
                                    ! for single message
else
  targetID = 0
  do i=1,ITER
    call MPI_Recv(buffer,N,MPI_BYTE,targetID,...)
    call MPI_Send(buffer,N,MPI_BYTE,targetID,...)
  enddo
endif
```

最主要的是，为了得到精确的测量时间，即使是短消息，乒乓消息传输也会重复几次（ITER）。记住这一“特殊性”，现在可以用它来解释带宽“驼峰”（见图 10-19）：进程 0 的 $sendb_0$ 到进程 1 的 $recvb_1$ 之间的传输可以在接收端作为一个单次复制操作来实现，即进程 1 执行 $recvb_1(1{:}N)=sendb_0(1{:}N)$，$N$ 为消息中的字节数。如果 N 足够小，那么来自 $sendb_0$ 中
256 的数据将位于进程 1 的 cache 中，并且不需要替换或者修改这些 cache 项除非 $sendb_0$ 发生更改。然而，发送缓冲区在每个进程的循环内核内不发生改变。因此，第一次迭代后发送缓冲区位于接收进程的 cache 内，同时 cache 内复制操作代替了在内存和超传输网络间的数据传输，而发生在随后的迭代中。

当消息较大时，性能下降的原因有两个：首先，L3 级 cache（2MB）对于容纳局部接收缓冲区和远程发送缓冲区或者两者中的一个都显得太小了。其次，IMB 的执行使得随消息量的增大，迭代次数在逐渐下降甚至当消息足够大时只完成一次迭代，即网络中最初的复制操作。



1. 第一次“乒”：P1 复制 $sendb_0$ 到 $recvb_1$，后者驻留在它的高速缓存里。

2. 第一次“乓”：P0 复制 $sendb_1$ 到 $recvb_0$，后者驻留在它的高速缓存里。

3. 第二次“乒”：P1 在它未经修改的 $recvb_1$ 上执行高速缓存内复制操作。

4. 第二次“乓”：P0 在它未经修改的 $recvb_0$ 上执行高速缓存内复制操作。

5. .... 在高速缓存内重复步骤 3 和 4。

图 10-19　当消息符合缓存大小时，共享内存系统内标准 MPI 乒乓事件链。C0 和 C1 分别指代处理器 P0 和 P1 的 cache。M0 和 M1 是 P0 和 P1 的局部内存

现实中的应用明显不能利用“性能驼峰”。为了评估那些能从大消息单次复制中获益的代码的内点通信的真实潜能，应该在内迭代中再增加一个数组 $sendb_i$ 和 $recvb_i$ 互换的乒乓操作（即指定 $sendb_i$ 为进程 i 上第二个 MPI_Recv() 的接收缓冲区），发送进程 i 再次获得 $sendb_i$ 的专有所属权。另一个替代方法是使用“循环缓冲区”，这时乒乓发送 / 接收对各自使用一个来自于更大的发送与接收缓冲区的小的滑动窗口。每次乒乓操作后移动窗口自己的大小，因此发送和接收缓冲区在内存中的位置是不断变化的。如果数组的大小比任何 cache 都大，那么即使一个单独的消息大小适合 cache 并且 MPI 库使用单次复制传输，也能确保所有的发送缓冲区是被驱逐到内存中的同一点。IMB 基准测试允许通过一个命令行选项而使用循环缓冲区，结果的性能数据（图 10-18 中的方形）显示没有超过缓存内消息大小。

有趣的是，内点和节点间带宽在大消息的情形下大致达到了同样的渐近性能，反驳了内点间点对点通信无限快的普遍误解。尽管这里展示的是一个具体的系统结构及软件环境，但这一观测结论在很多当代（混合）并行系统上具有普遍性，尤其对于“商业”集群更是如此。然而它们在具体细节上有很大变化，因此 MPI 库随时间不断进化，特性也随之变化。在图 10-21 和图 10-22 中我们展示了在一个由双插槽 Intel Xeon 5160 节点组成（参见图 10-20）、通过 DDR 极速互联连接的集群上的乒乓性能数据。两个图上的唯一差别是所使用的 MPI 库的版本号（比较 IntelMPI 3.0 和 3.1）。对 MPI 实现的实际修改细节被隐藏起来，但是对两个版本大的性能变化的观察显示内点通信的简单模型是存在问题的，可能会导致错误的结论。

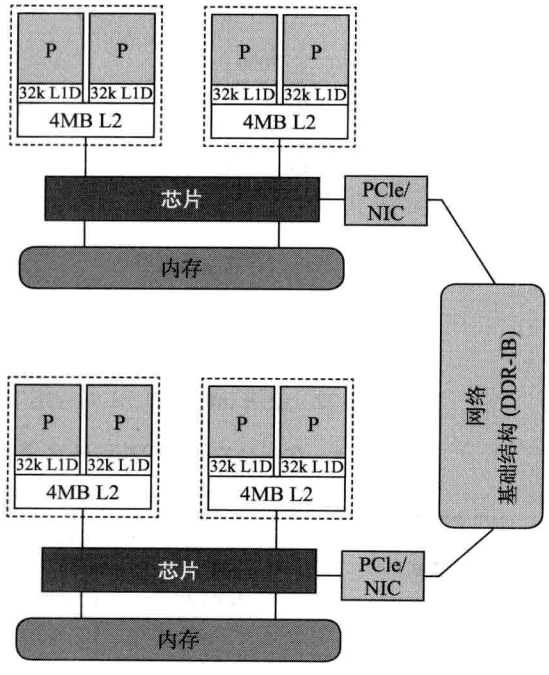


图 10-20　Xeon 5160 双插槽集群系统的两个节点，带有 DDR 极速互联

257

在小消息中，MPI 通信受延迟主导。对于上面描述的系统，IMB 乒乓基准测试衡量的延迟同渐近的带宽数目一起在表 10-1 中展示。很明显，当进程运行在同一节点上时延迟更小（如果它们共享 cache 还会更小）。我们强调这些基准测试在内点与节点间消息传输问题上只能给出一个粗糙的描述。如果多个进程对同时进行通信（现实应用经常如此），情况将变得更加复杂。更详细的分析见参考文献 [O72] 中混合 MPI/OpenMP 编程内容。

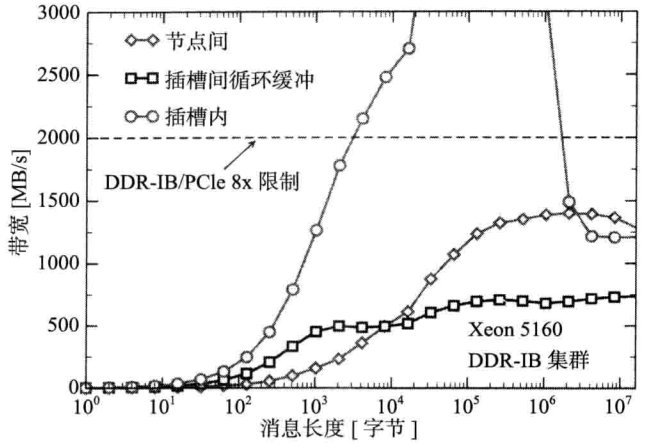


图 10-21 在一个 Xeon 5160 DDR-IB 集群上，使用 Intel MPI 3.0 的节点间、内点插槽间以及纯插槽内通信的 IMB 乒乓性能

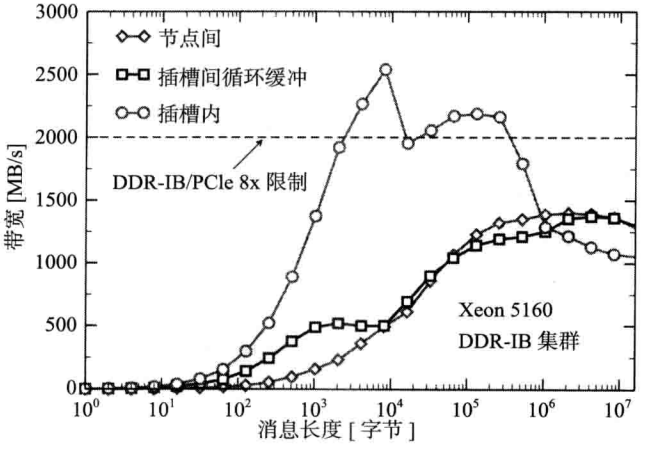


图 10-22 使用 Intel MPI 3.1 时与图 10-21 相同的基准测试。内点行为变化显著

表 10-1 在一台 Cray XT5 和一台带有 DDR 极速互联的商用 Xeon 集群上进行 IMB 乒乓基准测试得到的测量延迟及渐进带宽

模式	延迟 [μs]			带宽 [MB/s]		
	XT5	Xeon-IB		XT5	Xeon-IB	
	MPT3.1	IMPI3.0	IMPI3.1	MPT3.1	IMPI3.0	IMPI3.1
节点间	7.40	3.13	3.24	1500	1300	1300
插槽间	0.63	0.76	0.55	1400	750	1300
插槽内	0.49	0.40	0.31	1500	1200	1100

258 ~ 259

从上面展示的带宽和延迟特征中必须得到的最重要结论是进程 – 核的亲密性对在当今流

行的“各向异性”多插槽多核系统上的应用性能有重要影响（尽管不直接涉及通信，但如 6.2 节和 7.2.2 节中展示的那样，类似的效果也出现在 OpenMP 编程中）。因此 10.4.1 节中描述的映射问题与内点拓扑级别也变得相关；例如，给定适当的消息大小并且只要使用相邻点通信的 MPI 代码，相邻 MPI 进程应该位于同样的 cache 组中。当然，其他因素的约束如到内存和 NUMA 的共享数据路径也应该被考虑在内，这没有一个普遍的原则。还需要注意的是，在强扩展情形下可能会发生随处理器数的增加乒乓曲线下降到延迟驱动的状态，除非进程 / 线程的布局基于小数量的进程（参加习题 10.5）。

习题

10.1 *归约和竞争*。对比图 10-15 和图 10-16，能否想出一个网络拓扑结构使得在两个例子上的归约操作具有同样的性能？假定一个完全非阻塞的胖树网络，有什么其他因素会阻碍分层归约产生最优的性能？

10.2 *优化的全局归约*。我们将 MPI_Allreduce() 表述为 MPI_Reduce() 和 MPI_Bcast() 的结合。尽管这在语义上是正确的，但是这种方式实现 MPI_Allreduce() 的效率是很低的。怎样才能做得更好？

10.3 *急切与集合*。回顾 5.2 节中的并行化方法，在哪种典型情况下使用 MPI 的“急切”消息传输协议会产生副作用？可能的解决方案是什么？

10.4 *立方体是否总是最优的*？在 10.4.1 节中我们展示了如果区域沿着所有三个坐标方向切分，晕环交换带来的强可扩展通信开销展示出对工作进程数最有利的依赖性。请问这种策略下的通信开销总是最小的吗？

10.5 *乘坐乒乓曲线*。对于 10.4.1 节中展示的带有晕环交换的强扩展和立方体区域分解，导出一个有效带宽 B_{eff}（N，L，w,T_l,B）的表达式。假设一个点对点消息传输可以用简单的延迟 / 带宽模型（见 4.2 节）描述，并且在不同方向上的通信以及计算与通信间均无重叠。 260

10.6 *再论非阻塞 Jacobi*。在 9.3 节中，我们使用非阻塞接收来避免晕环交换的死锁。然而，在任何时刻，每个进程确切地有一个非阻塞请求是未完成的。请问代码可以重构成使用多个未完成请求吗？这样会有什么缺点吗？

10.7 *结合的发送与接收*。MPI_Sendrecv() 将标准发送（MPI_Send()）和标准接收（MPI_Recv）结合在一次调用中：

```
<type> sendbuf(*), recvbuf(*)
integer :: sendcount, sendtype, dest, sendtag,
           recvcount, recvtype, source, recvtag,
           comm, status(MPI_STATUS_SIZE), ierror
call MPI_Sendrecv(sendbuf,      ! send buffer
                  sendcount,    ! # of items to send
                  sendtype,     ! send data type
                  dest,         ! destination rank
                  sendtag,      ! tag for receive
                  recvbuf,      ! receive buffer
                  recvcount,    ! # of items to receive
                  recvtype,     ! recv data type
                  source,       ! source rank
                  recvtag,      ! tag for send
                  status,       ! status array for recv
                  comm,         ! communicator
                  ierror)       ! return value
```

你怎样实现这个函数以确保它在环形转换通信模式下不会产生死锁？会有其他的积极影响吗？

10.8 *负载均衡及区域分解*。在开边界（即非环形）条件下的三维（立方体）区域分解中，负载均衡上的隐含通信开销是什么？假设在整个并行系统中 MPI 通信性质是不变的、各向同性的，并且通信不能和计算重叠。为了弥补减少的表面积而增大最外层子域会有效果吗？ 261 ~ 262

第 11 章

Introduction to High Performance Computing for Scientists and Engineers

MPI 与 OpenMP 混合编程

当今的大规模并行计算机在整体系统级别使用专门的分布式存储，但在作为基本组成块的计算节点内使用共享内存。尽管这些混合架构已经使用了超过十年，但大多数并行应用仍然没有注意到硬件架构而只使用 MPI 实现并行化。如果人们考虑到大多数并行应用、解法器及方法的根基同 MPI 库本身一样可以追溯到“大”机器（例如著名的 Cray T3D/T3E MMP 系列）是纯粹的分布式存储的时代，这就不再是一件奇怪的事。随后存在的 MPI 应用和库就很容易传输到共享内存系统，因此大多数努力被花费在提升 MPI 的可扩展性上。此外，应用开发者们也委托 MPI 提供者开发高效 MPI 实现，以期能完全利用共享内存系统的能力用于高性能内点消息传输（与内点 MPI 相关的一些问题参见 10.5 节）。因此纯 MPI 隐含地假定为同节点间通信用 MPI、节点内并行用 OpenMP 的优化的、良好的混合 MPI/OpenMPI 代码一样高效。近些年来共享内存节点由小节点发展到中等大小（每个节点上不超过两个或四个处理器）的过程也帮助人们建立起一个常识，即对于相同的问题，混合代码性能通常不会优于只用 MPI 实现的版本。

在多核处理器时代，在每个核上运行一个 MPI 进程是否合适值得商榷。单一芯片的并行性将会稳定增长，同时共享内存节点将会有高度并行、分层次以及多核多插槽结构。本章将会指明这种发展，同时介绍在这种新的共享内存节点上编写及运行好的混合代码的基本原则。首先，我们将会讨论混合 OpenMP/MPI 编程可预料的优缺点。转到映射问题，我们将会指出混合性能同纯 MPI 代码一样，主要依赖的因素并不直接与编程模型相关，而与线程、进程到核之间关联相关。此外，一个节点内 OpenMP 线程与 MPI 进程能够如何精确地相互
263 作用会有几种选择，这给大多数混合应用留下了大量的提升空间。

11.1 基本 MPI/OpenMP 混合编程模型

混合 OpenMP/MPI 编程模型的基本想法是允许任意 MPI 进程以在纯 OpenMP 程序中发起主线程同样的方式发起一组 OpenMP 线程。因此，将 OpenMP 编译器指令插入到已存在的 MPI 代码中是构建第一个混合并行程序的一种直接方法。按照良好 OpenMP 编程指南，在质朴的混合代码中计算密集型循环结构是 OpenMP 并行化的首要目标。在发起 MPI 进程前，同一个纯粹的 OpenMP 程序一样，人们不得不指定每个 MPI 进程中 OpenMP 线程的最大数量。执行时，当每个 MPI 进程遇到一个 OpenMP 并行化的区域便激活一组线程（自己成为主线程）。

从纯 MPI 转换到混合执行的时候，MPI 进程间没有自动的同步，即在给定时刻，一些 MPI 进程有可能运行在完全不同的 OpenMP 并行区域内，同时其他进程位于程序的纯 MPI 部分。MPI 进程间的同步仍然受限于恰当 MPI 调用的使用。

我们定义两种基本的混合编程方法 [O69]：向量模式和任务模式。它们在 MPI 调用和 OpenMP 指令相互作用这个级别上有所区别。下面将使用并行三维 Jacobi 解法器作为例子简

要介绍两种方法的基本思想。

11.1.1 向量模式实现

在向量模式的实现中，所有 MPI 子例程位于 OpenMP 并行区域外被调用，即在 OpenMP 代码中的“串行”部分。这种方法的主要优点是易于编程，因为一个已有的纯 MPI 代码可以通过在耗时的循环前增加 OpenMP 工作共享指令以及注意适当的 NUMA 放置（见第 8 章）转换成混合代码。一个向量模式实现 3 维 Jacobi 解法器核心的伪代码展示在代码清单 11-1 中。这与 9.3 节中展出的纯 MPI 并行化版本十分相似，而且的确在 MPI 层和 OpenMP 指令间不存在干扰。编程遵循各种独立的范式指南。向量模式策略与使用 MPI 编程的并行向量计算机相似，后者在并行的内部实现上利用了向量化及多通道流水线。典型的可以从这种模式中获益的例子是那种 MPI 进程数受限于指定问题约束的应用。通过多线程利用额外（更低级别的）细粒度级别是超出 MPI 限制而增加并行性的唯一方法 [O70]。 264

代码清单 11-1 一个向量模式下 3 维 Jacobi 解法器混合实现的伪代码

```
1   do iteration=1,MAXITER
2   ...
3  !$OMP PARALLEL DO PRIVATE(..)
4     do k = 1,N
5  ! Standard 3D Jacobi iteration here
6  ! updating all cells
7        ...
8     enddo
9  !$OMP END PARALLEL DO
10
11 ! halo exchange
12    ...
13    do dir=i,j,k
14
15      call MPI_Irecv( halo data from neighbor in -dir direction )
16      call MPI_Isend( data to neighbor in +dir direction )
17
18      call MPI_Irecv( halo data from neighbor in +dir direction )
19      call MPI_Isend( data to neighbor in -dir direction )
20    enddo
21    call MPI_Waitall( )
22  enddo
```

11.1.2 任务模式实现

任务模式十分普遍并且允许在 OpenMP 并行区域内进行任意种类的 MPI 通信。基于消息传递库的线程安全需求，MPI 标准在 OpenMP 和 MPI 间定义了三个不同级别的冲突（参见下面的 11.2 节）。在使用任务模式之前，代码必须检查 MPI 库支持哪种级别。功能性任务分解和通信与计算的解耦合是任务模式可能会派上用场的两个地方。作为后者的一个例子，三维 Jacobi 解法器核的任务模式实现概述在代码清单 11-2 中。这里主线程负责更新边界单元（6 ～ 9 行），即局部过程子区域的表面单元以及与相邻进程交换更新值（13 ～ 20 行）。这些可以伴随着由剩余线程（24 ～ 40）执行的所有内部单元更新而同时完成。在一次完全更新区域后，需要对每个进程内的所有 OpenMP 线程同步，同时 MPI 同步只在需要晕环交换的最近相邻进程间间接发生。

任务模式具有很高的灵活性，但是同时也使代码变得膨胀并且极大地增加了编程的复杂性。一个主要的问题是 MPI 和 OpenMP 都没有内嵌的机制来直接支持任务模式方法。因

此，人们通常也在 OpenMP 级别上以 MPI 风格的编程结尾。不同的功能任务需要映射到
265 OpenMP 线程标识上（第 5 行开始的 if 语句），同时可能会执行在代码的不同部分。

代码清单 11-2 一个任务模式下 3D Jacobi 解法器混合实现的伪代码

```
1  !$OMP PARALLEL PRIVATE(iteration,threadID,k,j,i,...)
2    threadID  = omp_get_thread_num()
3    do iteration=1,MAXITER
4    ...
5    if(threadID .eq. 0) then
6    ...
7  ! Standard 3D Jacobi iteration
8  ! updating BOUNDARY cells
9    ...
10 ! After updating BOUNDARY cells
11 ! do halo exchange
12
13     do dir=i,j,k
14       call MPI_Irecv( halo data from neighbor in -dir direction )
15       call MPI_Send( data to neighbor in +dir direction )
16       call MPI_Irecv( halo data from neighbor in +dir direction )
17       call MPI_Send( data to neighbor in -dir direction )
18     enddo
19
20     call MPI_Waitall( )
21
22   else ! not thread ID 0
23
24 ! Remaining threads perform
25 ! update of INNER cells 2,...,N-1
26 ! Distribute outer loop iterations manually:
27
28     chunksize = (N-2) / (omp_get_num_threads()-1) + 1
29     my_k_start = 2 + (threadID-1)*chunksize
30     my_k_end = 2 + (threadID-1+1)*chunksize-1
31     my_k_end = min(my_k_end, (N-2))
32
33 ! INNER cell updates
34     do k = my_k_start , my_k_end
35       do j = 2, (N-1)
36         do i = 2, (N-1)
37           ...
38         enddo
39       enddo
40     enddo
41   endif ! thread ID
42 !$OMP BARRIER
43   enddo
266 44 !$OMP END PARALLEL
```

因此，方便的 OpenMP 工作共享并行化指令不能再被使用。工作负载不得不在更新内部单元的线程间手动分配（28 ～ 31 行）。注意，到目前为止我们只提出了任务模式模型中最简单的一种。依赖于 MPI 库对线程级别的支持，人们也可以在一个 OpenMP 并行区域的所有线程上使用 MPI 调用，这将进一步阻碍可编程性。最后必须强调的是这种混合方法阻止了增量混合并行化，因为一个已有 MPI 代码的实质部分需要完全重写。这也将一个纯 MPI 及单一代码中混合版本的可维护性严重复杂化了。

11.1.3 案例分析：混合 Jacobi 解法器

在 9.3 节中我们开发了 MPI- 并行三维 Jacobi 解法器，它是一个评估混合编程在实际应

用情形下潜在益处的很好例子。为了证实前面章节的讨论，我们现在对比向量模式和任务模式的实现。图 11-1 用 MLUP/s 展示了两种混合版本以及使用两种不同标准网络（千兆位以太网和 DDR 极速互联）的纯 MPI 变体性能数据。为了最小化亲缘性及位置效应（对比接下来部分的广泛讨论），我们选取了同 9.3.2 节中纯 MPI 例子一样的单插槽双核集群。然而，每个节点上的两个核自始至终都在使用，因而与图 9-11 相比极速互联的整体性能提高了 10% ～ 15%（对于 480^3 的同样大小区域）。由于在使用极速互联时通信开销仍旧几乎可以忽略，因此纯 MPI 变体扩展性非常好。混合编程没有展示出获益并不令人奇怪，因为极速互联网络与所有的三种变体（虚线）获得了相同的性能水平。

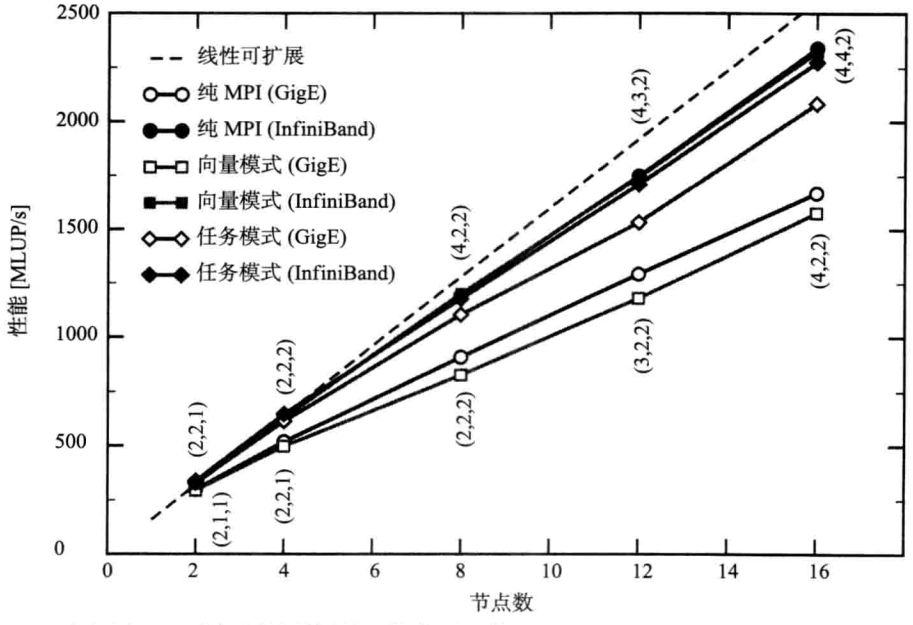


图 11-1 在与图 9-10 中相同的单插槽双核集群上使用 DDR 极速互联（实心符号）与千兆位以太网（空心符号）对比的三维 Jacobi 解法器强可扩展性（问题大小 480^3）的纯 MPI(圆圈）和混合版本（方形和钻形）的并行性能。区域分解拓扑（每个笛卡儿方向上的进程数）在每个数据点上表示出来（低于混合版本但高于纯 MPI 版本）。细节参见正文

对于千兆位以太网，图形完全发生了改变。甚至在节点数较少时，数据交换的成本大幅度降低了纯 MPI 版本的并行可扩展性，并且在千兆位以太网和极速互联之间有日益增长的性能差距。此时向量模式毫无帮助，因为计算和通信仍然是串行的；相反性能甚至会下降，因为在 MPI 通信期间每个节点上只有一个核是活跃的。然而在任务模式下，通信和计算的重叠是很有可能的。在这种情况下，并行的可扩展性显著提高，同时千兆位以太网的性能接近于极速互联水平。

对这个简单例子的研究揭示了混合编程的最重要规则：只有在纯 MPI 的可扩展性不令人满意时考虑尝试混合。在一个混合实现上下功夫并且尝试比一个完美扩展的 MPI 代码更快是毫无意义的。 267

11.2 MPI 线程交互分类

从单线程转换到多线程执行可不仅仅如一个通信库视角那样是一件简单的事情。一个

MPI 库的完全“线程安全”实现是一件困难的任务。提供共享的连续消息队列或者连续的内部消息缓冲区是这里提到的两个挑战。对于 MPI 库最灵活同时也最糟糕的情形是 MPI 通信允许在任意时刻的任意线程上发生。由于 MPI 可能实现在很少甚至没有线程支持的环境下，所以现行的 MPI 标准（版本 2.2）区分 4 种不同级别的线程互操作，从没有线程支持开始（“MPI_THREAD_SINGLE”）到最一般的情形（“MPI_THREAD_MULTIPLE”）：

- MPI_THREAD_SINGLE：只有一个线程会执行。
- MPI_THREAD_FUNNELED：进程可能是多线程的，但只有主线程会使用 MPI 调用。
- MPI_THREAD_SERIALIZED：进程可能是多线程的，多个线程都可以使用 MPI 调用，但一次只有一个；MPI 调用不能同时被两个不同的线程使用。
- MPI_THREAD_MULTIPLE：多个线程在任意时刻均可调用 MPI，没有任何限制。

每个混合代码应该总是使用 MPI_Init_thread() 函数调用来检查所需的线程支持的级别。图 11-2 提供了一个被 MPI 线程级互操作允许的不同混合 MPI/OpenMP 模式的概述图。上面提到的并行三维 Jacobi 解法器的两种混合实现都需要 MPI 对 MPI_THREAD_FUNNELED 的支持，因为主线程是唯一发起 MPI 调用的线程。任务模式版本也提供了对由 MPI 多线程执行引起的并发症的第一个见解。更重要的是，MPI 不允许在同一进程内的不同线程间显式寻址。如果在线程和 MPI 调用间有一个强制映射，那么程序员将不得不实现它。这可以通过显式地使用 OpenMP 线程号和潜在地将它们与不同消息标签连接来实现（即相同 MPI 进程的不同线程的消息由唯一的 MPI 消息标签区分）。另一个需要注意的问题是同步 MPI 调用仅阻塞调用的线程，如果可能的话，会允许相同 MPI 进程的其他线程执行。还有一些更重

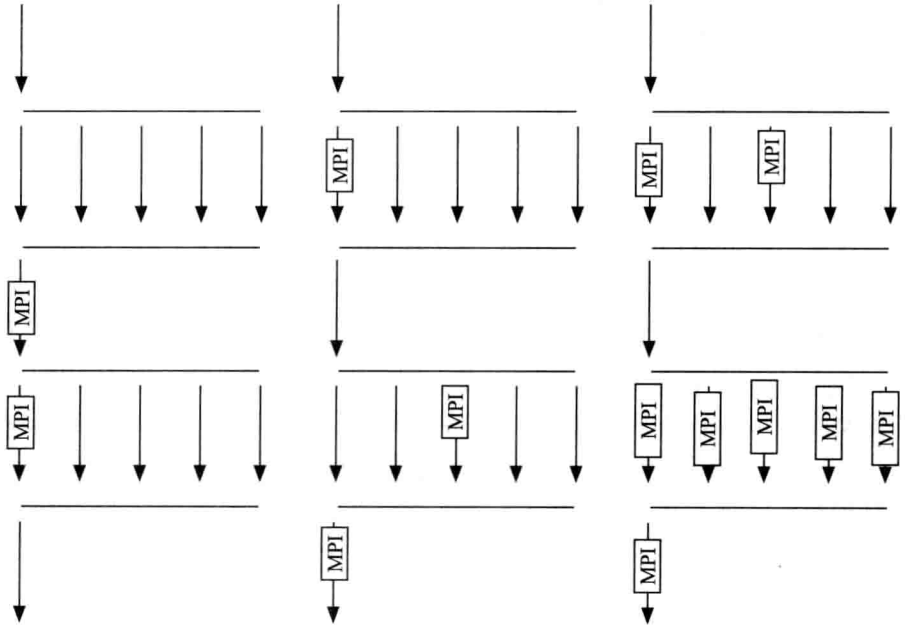


图 11-2 MPI 支持的线程级别：a）MPI_THREAD_FUNNELED，b）MPI_THREAD_SERIALIZED，c）MPI_THREAD_MULTIPLE。简单 MPI 模式 MPI_THREAD_SINGLE 被省略了。各种级别的线程支持允许的典型通信模型被描述为拥有很多 OpenMP 线程的单一多线程 MPI 进程

要的指南需要考虑，尤其在需要完全利用 MPI_THREAD_MULTIPLE 功能时。在编写多线程 MPI 代码时，通读 MPI 标准文档 [P15] 中的“MPI 和线程”章节是必需的。

11.3 混合分解及映射

一旦已经实现一个混合 OpenMP/MPI 代码并且也已申请计算资源申请，那么在启动应用程序之前需要做两个重要的决策。

首先，人们需要选择每个 MPI 进程上 OpenMP 线程的数量以及每个节点上的 MPI 进程数。当然手头的共享内存节点的能力对选择强加了一些限制；例如，每个节点上的线程总数不应该超出计算节点的核数。在一些罕见例子中，每个虚核上运行一个单线程（如果处理器高效地支持并发多线程，见 1.5 节），或者甚至使用比可用核数更少的线程（如果每个线程上的内存带宽或者 cache 大小是一个瓶颈）会是有益处的。此外，待解决的物理问题以及底层硬件架构也强烈地影响着混合分解的最优选择。

MPI 进程和 OpenMP 线程间到同一个计算节点上的插槽和核上的映射是另一个重要的决策。关于这点，基本的节点特征（插口数量以及每个插口上的核数）可以被用作第一准则，但即使是在装有多核处理器芯片的相当简单的双插槽计算节点上也会有一个与分解和映射选择相关的大的参数空间。在图 11-3 ～图 11-6 中，一个具有代表性的子集被描述为由两个标准双插槽、四核 ccNUMA 节点组成的集群。此处，我们隐含 MPI 库支持 MPI_THREAD_FUNNELED 级别，即每个 MPI 进程的主进程（t_0）假定处于一个突出的位置。这对于上面提到的两个例子均是有效的并且影响在很多混合应用中实现的方法。

11.3.1 每个节点一个 MPI 进程

仅考虑共享内存特征，人们可以简单地在每个节点上指派一个单一的 MPI 进程（m_0，m_1），同时发起八个 OpenMP 线程（t_0,...,t_7），比如每个核上一个线程（见图 11-3）。这在硬件设计和混合分解上有一个明显的不对称，这一点会在一些性能相关的问题上展现出来。全部线程进行同步代价昂贵，因为它包括片外数据交换，并且它在多插槽和多核设计 [M41] 时会成为一个主要的瓶颈（如何估计 OpenMP 中的同步开销参见 7.2.2 节）。在实现代码时，NUMA 优化（见第 8 章）需要被考虑在内；特殊情形下，如果 MPI 进程（比如运行在 LD0） 270
分配实质性的消息缓冲区，那么典型的局部性和竞争问题可能会出现。此外，主线程在为 MPI 调用收集数据时会产生非局部数据访问。如果可用的节点间带宽在单一 MPI 进程下不能达到饱和，那么即使使用性能较低的核，每个节点上使用单一 MPI 进程也是不能充分利用最先进的互连技术的 [O69]。缓解发起 MPI 进程并阻塞线程同将 MPI 进程数归约到最小值一样是这个简单混合分解模型的典型优势。

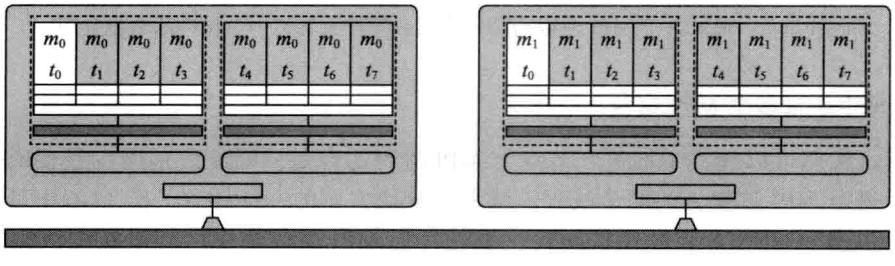


图 11-3 映射携带 8 个线程的单一 MPI 进程到每个节点

11.3.2 每个插槽一个 MPI 进程

分配一个多线程 MPI 进程到每个插槽完美地匹配了节点拓扑结构（参见图 11-4）。然而，正确地发起 MPI 进程同时顺时针阻塞 OpenMP 线程到插槽上需要十分的谨慎。现在，MPI 通信将会同时发生在插槽间和节点间，并且在应用中应该考虑适当调度 MPI 调用来重叠插槽间和节点间通信 [O72]。另一方面，由于每个 MPI 进程都限制在了一个单独的局部区域，因而这种映射避免了 ccNUMA 数据局部性问题。同时每个 MPI 进程上的所有线程对于单一共享 cache 的访问允许快速线程同步，增加了线程间 cache 重用的概率。注意，这样的讨论结果需要被推广到带有共享的片外级别缓存的核组上：对于 Intel 四核芯片的第一代，双核核组共享一个 L2 级缓存，无可用的 L3 缓存。对于这种芯片架构，人们应该在每个 L2 级 cache 组上使用一个 MPI 进程，即每个插槽上两个进程。

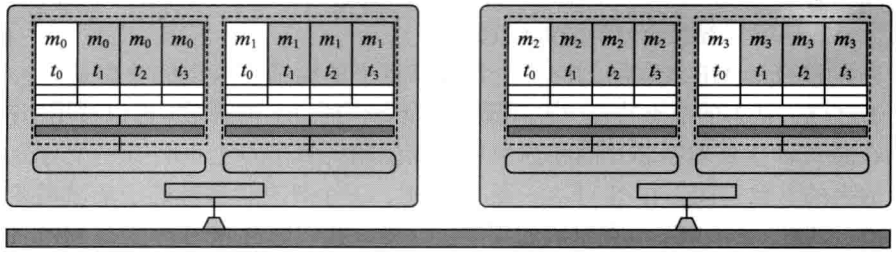


图 11-4 映射携带 4 个线程的 MPI 进程到每个插槽（L3 组或者局部区域）

关于映射的一点小的修改能够容易地出现在完全不同的场景中，此时既不改变每个节点上的 MPI 进程数，也不改变每个进程上的 OpenMP 线程数：例如，如果在插槽间选择了循环线程分布，将会出现图 11-5 中展示的情形。在每个节点上，每个插槽持有两个 MPI 进程，潜在地允许在它们之间通过共享 cache 进行快速通信。然而，每个插槽上不同 MPI 进程的线程是相互交错的，这使得高效 ccNUMA 编程自身成为一个挑战。此外，一个完全不同的负载特性被指派到同一节点上的两个插槽。从线程同步和远程数据访问的角度，这一切接近于一个最坏的情形。

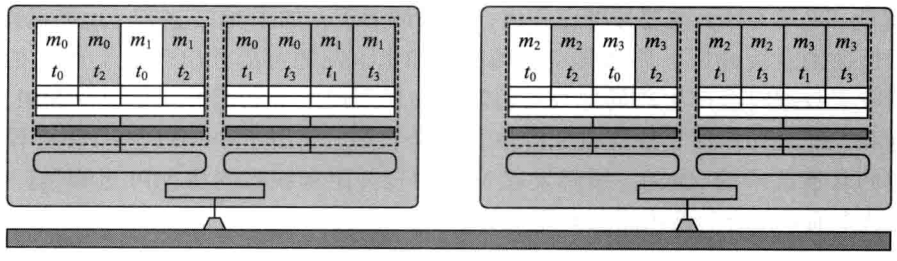


图 11-5 映射两个 MPI 进程到每个节点上同时实现了一个循环的线程分布

11.3.3 每个插槽多个 MPI 进程

当然，人们可以继续增加每个节点上 MPI 进程的数量同时相应地减少线程的数量，最后终止于每个 MPI 进程上的两个线程。线程在分块方面的选择导致了图 11-6 中呈现出的有利情形。虽然 ccNUMA 问题在这里是次要的，但 MPI 通信会在所有潜在的级别上展现出来：插槽内、插槽间以及节点间。因此，以最小化最慢通信路径访问的方式将计算区域映射

271
~
272

到 MPI 进程是一种潜在的优化策略。这种分解方法对受限于 MPI 可扩展性的内存密集型代码也是有益的。通常一半的线程数已经能够使单一插槽上的主内存带宽达到饱和，使用携带多线程的 MPI 进程能从数量较少的 MPI 进程中获得一点性能增加。作为一种选择，功能分解可以被应用于显示地隐藏 MPI 通信（参见上面描述的 Jacobi 解法器的任务模式实现）。很明显，对于这个分解，大量不同的映射策略也是可用的。然而，由于上面多次强调的对称参数，图 11-6 中的映射通常会提供最好的性能，因此应该被最先测试。

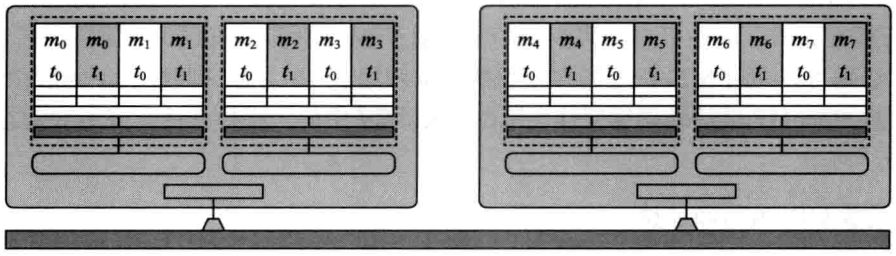


图 11-6 映射两个 MPI 进程到一个单一插槽，其中每个进程有两个线程

不幸的是，在编写时大多数 MPI 实现对于规定到各自核上不同的映射策略以及正确的阻塞线程的支持少得可怜。即便存在一些支持，通常也被限定于 MPI 和编译器的特定组合。因此，正确地发起混合应用更多的是程序员的责任，但是对于混合代码预先的性能研究是不可或缺的。有关映射和亲和力问题的更多信息参见 10.4.1 节及附录 A。

11.4 混合编程的优势和劣势

由于较早提到的原因，混合并行 MPI/OpenMP 方法在并行应用中仍然很少能最大限度地实现。因此，没有完整的可用理论保证混合方法对于代码重构或者从头设计一个完全混合的应用能够抵偿额外的成本便不再让人感到惊讶。然而到目前为止，已经确定了混合方法相对于纯使用 MPI 的一些潜在的基本优势和劣势。下面，我们简要总结它们中最重要的几个。以下每个主题的影响很大程度上将会依赖于特定的应用代码甚至是输入数据的选择。当且仅当纯 MPI 代码的可扩展行不能提供令人满意的并行性能时，对那些问题的仔细调查是必需的。

11.4.1 改善的收敛速度

很多迭代解法器包含循环间的数据依赖性，例如，著名的经典 Gauss-Seidel 格式（参见 6.3 节）。如果标准的区域分解用于 MPI 并行化，那些依赖性会在边界单元上打破。虽然算法仍然收敛到正确的稳态解，但收敛速度将会随着子区域数量的增加而下降（对于强扩展情形）。此处，一个混合方法会帮助减少子区域的数量同时提高收敛速度。例如在 [A90] 的一个带有隐式解法器的 CFD 应用中，这种现象被展示出来：在每个节点上只发起一个 MPI 进程（在一个子域内计算），同时在节点内部使用 OpenMP，这种方法加速了并行算法的收敛。这个例子清楚地表明并行加速比或者整体浮点性能是量化“混合红利”的错误衡量方法。解 273
的运行时间则是一种合适的替代方案。

11.4.2 共享 cache 中的数据重用

对单一共享 cache 上线程操作使用共享内存编程的模型极大地扩展了优化机会：对于具有常规模板依赖性的迭代解法器，如 Jacobi 或者 Gauss-Seidel 类型，被第一个线程导入或修

改的数据可以被 cache 中另一个线程读取，并且在被驱逐到主存前再次更新。这个技巧导致了一个高效且自然的并行临时阻塞，但需要一个共享地址空间来避免缓存数据和冗余数据复制的双缓冲（正如将要被一个纯 MPI 实现强制执行那样）[O52，O53，O63]。为了在大规模并行代码中实现这样供选择的多核方面策略，混合并行化是强制性的。

11.4.3 利用额外级别的并行性

很多计算任务提供了一个问题内在的粗粒度并行级别。一个突出的例子是一些多层 NAS 并行基准测试，其中只有相当少量的区域对于 MPI 并行是可用的（从几十到 256），额外的并行级别可以通过 MPI 进程的多线程执行来利用 [O70]。在这些级别上的潜在负载均衡也可以通过 OpenMP 灵活地调度变体（如在 OpenMPI 中的“指导性的”或者“动态的”）来高效解决。

11.4.4 重叠 MPI 通信和计算

MPI 提供了借助非阻塞 MPI 通信重叠通信和计算的灵活性，即先进行一些计算，随后检验 MPI 调用是否完成。然而，正如 10.4.3 节中描绘的那样，即使在使用非阻塞调用的情况下，今天的大部分 MPI 库并不执行真正的异步传输。如果 MPI 只在库代码执行的时候才向前推进（完全符合 MPI 标准），那么消息传递开销和计算实际上将被串行化。作为补救，程序员在每个 MPI 进程上使用单一线程以异步地执行 MPI 通信。具体例子见上面的 11.1.3 节。

11.4.5 减少 MPI 开销

MPI 通信开销对整体运行时间的贡献会随着 MPI 进程数的增加而迅速增加，尤其对于强扩展情形。同样，区域分解方法可以作为一个范式。局部子区域上表面积（通信）和体积（计算）的比率随着进程数的增加变得更糟（见 10.4 节），与此同时平均的消息大小变小。这
274 也可以降低有效通信带宽（参见习题 10.5）。同样，用于晕环层交换的整体缓冲区的大小也随 MPI 进程数的增加而增大，因而在处理器数目很多时会占用相当大一部分的主存。通过混合编程减少 MPI 进程的数量会有助于增加 MPI 消息长度同时减少整体内存占用。

11.4.6 多级别开销

通常情况下，编写一个真正高效且可扩展的 OpenMP 程序是完全不容易的，尽管增量并行化方法表面上看似简单。在第 7 章中我们已经演示了一些潜在的缺陷。在一个更抽象的层次上，引入第二个并行级别到程序中也带来了一个新的层，在这个层上基本的可扩展限制，如 Amdahl 定律，必须被考虑。

11.4.7 向量模式下批量同步通信

在混合向量模式中，所有的 MPI 通信于 OpenMP 并行区域外发生。换言之，所有在多线程进程中进出的数据只在所有线程都已经同步后才开始传输：通信在节点级别上是批量同步的，单一线程在 MPI 向前推进的时候没有机会进行有用的工作（除了支持真正的异步消息传输情形；更多信息参见 10.4.3 节）。相反，共享网络连接的一些 MPI 进程可以使用它任意多次，这经常会导致计算和通信的自然重叠，尤其是在允许急切交付的情形下。即使纯 MPI
275 程序也是批量同步的（如 9.3 节中展示的 MPI- 并行 Jacobi 解法器），运行时不同进程间的细
~ 微变化也可能导致至少一部分的重叠。
276

多核环境中的拓扑和亲缘性

本书已经多次强调过亲缘性机制的重要性，例如在考虑cache大小、带宽瓶颈、OpenMP并行化开销、ccNUMA局部性、MPI节点内通信以及MPI/OpenMP混合编程效率时都需要考虑共享存储系统中线程和进程对核的绑定。一般情况下，需要考虑该问题的以下三个方面问题：

- 拓扑：与处理器体系结构和操作系统无关，所有系统使用一些编号方式，对每个硬件线程（如果SMT可用）、计算核和NUMA局部性域设置一个唯一的整数（或者一组整数）。局部性控制一些系统工具和库利用这些整数，所以需要了解编号方案。通常为了方便起见，系统中有一个抽象层，允许用户指定例如插槽、cache组等构件。在最低层次上我们利用核ID，这个数值可能表示一个真实的物理核，或者如果处理器支持超线程，该值也可能表示一组硬件线程。有时逻辑计算核也表示硬件线程。
- 线程亲缘性：在确定哪些线程或者进程需要被绑定之后，绑定操作必须通过程序自身（利用合适的库或者系统调用）或者外部工具进行显式设置。一些操作系统可以维护线程和处理器核间的强亲缘性，亦即一个线程（或者进程）将持续在初始被分配的处理器核上执行计算，但是其他系统进程或者交互式方法会改变线程所在的硬件核，这就不能保证初始分配的状态依然成立。线程亲缘性的一个指标为性能抖动数值（亦即不同配置下的性能差异）。
- NUMA配置：（内存亲缘性，memory affinity）ccNUMA系统中结合线程亲缘性使用8.1.1节提到的首次分配策略，但是有时需要更为细粒度的控制，例如动态调度中负载均衡的需求，可能需要轮询分配策略，和线程绑定一样，这可以通过程序控制或者一些独立工具完成。

代码清单 A-1　双路八核 Intel Nehalem 系统上 likwid-topology-g 的输出结果

```
1   ---------------------------------------------------------------
2   CPU name:       Intel Core i7 processor
3   CPU clock:      2666745374 Hz

5   ***************************************************************
6   Hardware Thread Topology
7   ***************************************************************
8   Sockets:                2
9   Cores per socket:       4
10  Threads per core:       2
11  ---------------------------------------------------------------
12  HWThread        Thread          Core            Socket
13  0               0               0               0
14  1               0               1               0
15  2               0               2               0
16  3               0               3               0
17  4               0               0               1
18  5               0               1               1
19  6               0               2               1
20  7               0               3               1
21  8               1               0               0
```

```
9               1               1               0
10              1               2               0
11              1               3               0
12              1               0               1
13              1               1               1
14              1               2               1
15              1               3               1
--------------------------------------------------------------------
Socket 0: ( 0 8 1 9 2 10 3 11 )
Socket 1: ( 4 12 5 13 6 14 7 15 )
--------------------------------------------------------------------

************************************************************
Cache Topology
************************************************************
Level:  1
Size:   32 kB
Cache groups: ( 0 8 ) ( 1 9 ) ( 2 10 ) ( 3 11 ) ( 4 12 ) ( 5 13 ) ( 6 14 ) ( 7 15 )
--------------------------------------------------------------------
Level:  2
Size:   256 kB
Cache groups: ( 0 8 ) ( 1 9 ) ( 2 10 ) ( 3 11 ) ( 4 12 ) ( 5 13 ) ( 6 14 ) ( 7 15 )
--------------------------------------------------------------------
Level:  3
Size:   8 MB
Cache groups: ( 0 8 1 9 2 10 3 11 ) ( 4 12 5 13 6 14 7 15 )
--------------------------------------------------------------------

************************************************************
Graphical:
************************************************************
Socket 0:
+---------------------------------+
| +-----+ +-----+ +-----+ +-----+ |
| | 0  8| | 1  9| |2  10| |3  11| |
| +-----+ +-----+ +-----+ +-----+ |
| +-----+ +-----+ +-----+ +-----+ |
| | 32kB| | 32kB| | 32kB| | 32kB| |
| +-----+ +-----+ +-----+ +-----+ |
| +-----+ +-----+ +-----+ +-----+ |
| |256kB| |256kB| |256kB| |256kB| |
| +-----+ +-----+ +-----+ +-----+ |
| +-----------------------------+ |
| |              8MB            | |
| +-----------------------------+ |
+---------------------------------+
Socket 1:
+---------------------------------+
| +-----+ +-----+ +-----+ +-----+ |
| |4  12| |5  13| |6  14| |7  15| |
| +-----+ +-----+ +-----+ +-----+ |
| +-----+ +-----+ +-----+ +-----+ |
| | 32kB| | 32kB| | 32kB| | 32kB| |
| +-----+ +-----+ +-----+ +-----+ |
| +-----+ +-----+ +-----+ +-----+ |
| |256kB| |256kB| |256kB| |256kB| |
| +-----+ +-----+ +-----+ +-----+ |
| +-----------------------------+ |
| |              8MB            | |
| +-----------------------------+ |
+---------------------------------+
```

278

许多用户甚至是应用程序员都有一个错误的观点，即操作系统负责处理这些事务，但是在高性能计算领域这些问题需要用户显式处理，但是困难之处在于同时处理拓扑、亲缘性和NUMA需要考虑系统特征。虽然有很多提供自动化亲缘性控制工具[T30]的尝试，但是我们认为现在仍然需要对系统拓扑结构进行深入了解并进行手工配置。

由于Linux系统在高性能计算集群领域的主流地位（TOP 500[W121]中Linux系统的份

额比例从 1998 年的 0% 增长到 2009 年的 89%），本章介绍 x86 Linux 系统上的工具和库。我们忽略编译器相关的亲缘性机制，因为这些内容已经在编译器手册中提供。

注意本章提供的工具和库是根据作者经验来选择的，读者可能需要不同的工具。

A.1 拓扑

多核拓扑是指基于多核的共享内存计算机和计算核、cache、插槽以及数据通路的逻辑配置。1.4 节给出了多核芯片上 cache 组的组织结构，4.2 节包含了构建共享储存计算机的不同方式，通常官方手册中提供了系统中多核芯片和 NUMA 域的详细组织方式。

然而，由于操作系统内核或者固件（BIOS）版本的不同，因此硬件映射核 ID 的方式也能随之改变，所以首先需要确定每个核在系统中的实际物理位置。命令“cat/proc/cpuinfo”的输出信息非常有限，因为它仅提供有限的 cache 信息。LIKWID 高性能计算工具集 [T20,W120] 中的 likwid-topology 命令可以提供更为全面的信息，表 A-1 显示了该命令在两路四核 Intel Nehalem（core i7 体系结构，SMT）节点上带有“-g”参数的输出，该工具显示了所有的硬件线程、物理核、cache 大小和插槽。在本例中，我们有两个插槽，每个处理器有四个核并且每个核有两个硬件线程（8 ～ 10 行）。13 ～ 28 行显示了物理核到逻辑核心的映射方式：硬件线程 k（$k \in \{0...7\}$）和 $k+8$ 属于一个相同的物理核，37 ～ 47 行显示了 cache 组及其大小：一个插槽内的所有核共享一个 L3 级 cache，每个核独享一个 L1 和 L2cache 组。从 53 行开始的最后这些内容总结了所有这些信息。带有“-c”参数的 likwid-topology 命令可以提供更为全面的 cache 组织信息，包括相联度、cache 行大小等（只针对 L3 级 cache）：

279

```
1    Level:   3
2    Size:    8 MB
3    Type: Unified cache
4    Associativity: 16
5    Number of sets: 8192
6    Cache line size: 64
7    Non Inclusive cache
8    Shared among 8 threads
9    Cache groups:    ( 0 8 1 9 2 10 3 11 ) ( 4 12 5 13 6 14 7 15 )
```

LIKWID 软件包支持现在的 Intel 和 AMD x86 处理器，此外，类似 2.1.2 节展示的内容，它还可以读取运行时相关的硬件计数器信息，并且显式设置线程和硬件核心的亲缘性（参考下面章节内容）。

A.2 线程和进程分布

由于操作系统不了解用户程序的性能特征，因此不能依赖默认的操作系统对于线程布局的配置。当了解硬件线程、核、cache 和插槽拓扑等信息之后，用户可以显式指定适用于并行应用程序性能特征的线程和核间的映射绑定关系。例如带宽受限型应用的线程分到不同处理器上可能会提升性能，相反，如果并行程序的多个 MPI 邻接进程间需要进行频繁同步操作，需要将它们尽量分配到相邻的硬件核心组上以便加速同步操作。除非显式指出，否则下面我们将不区别线程和进程两个术语。本节中所有实例的硬件平台为在 A.1 节中提到的集群系统。

A.2.1 外部亲缘性工具

如果不能改变应用程序代码，就只能利用外部工具配置程序亲缘性，并且由于基于代码的亲缘性不具备可移植性，因此这是常用的方法。Linux 操作系统提供 taskset 工具（属于 util-linux-ng 软件包的一部分），允许用户通过设置掩码来调整进程的亲缘性：

```
taskset [options] <mask> <command> [args]
```

掩码可以通过一组核 ID 列表或位模式来设置（需要在命令行中加 -c 选项），下面的例子限制了一个应用程序的线程为核 ID 0 ～ 7：

```
$ export OMP_NUM_THREADS=8
$ taskset -c 0-7 ./a.out        # alternative: taskset 0xFF ./a.out
```

280

需要确定所有的线程都在 8 个不同的核上运行，但是并没有绑定线程到核的映射关系，线程可以在掩码确定的处理器核心范围内迁移。虽然操作系统尽量阻止多个线程运行在同一个核上，但是映射关系仍然可能破坏 NUMA 局部性。同样，如果设置 OMP_NUM_THREADS=4，那么依然不能确定 8 个核中的哪 4 个核被利用。总之，taskset 工具更适用于绑定单线程进程。但是当一组线程需要运行于十分对称的环境时，例如一组 L2 级 cache，也可以使用 taskset。

在产品环境中，taskset 之类的工具应该应用于 MPI 启动进程（亦即 mpirun 或者其他类似命令）。

```
$ mpirun -npernode 8 taskset -c 0-7 ./a.out
```

选项 -npernode 8 指定每个节点启动 8 个进程。通过 taskset 的掩码设置 0xFF，每个 MPI 进程（a.out）只能在同一组 8 个不同物理核上运行，但是如前所述，进程可以在内的核上随意迁移。只有当默认映射能够提供较好性能时才不会降低 NUMA 局部性，并且只有当每个 taskset 外部命令均能正常配置 MPI 进程时这种方法才有效。

守护线程并不真正执行应用程序代码，因此不应该被绑定，所以外部亲缘性控制工具需要确定守护线程的位置和启动时间（依赖编译器）。幸运的是，Linux 下的 OpenMP 实现基于 POSIX 线程，并且 OpenMP 线程在首次进入并行区时通过调用 pthread_create() 函数启动，通过重载 pthread_create() 函数就可以自行配置守护进程 [T31]，这也适用于 MPI/OpenMP 混合程序，不过配置附加的 MPI 进程会增加复杂性。LIKWID 工具集 [T20，W120] 包含一个称为 likwid-pin 的轻量级程序，可以将进程中的某个线程绑定到一个节点的固定核上。但是如果想跳过守护进程，需要显式配置掩码。

```
likwid-pin -s <hex skip mask> -c <core list> <command> [args]
```

掩码中的第 b 位与第 b+1 个调用 pthread_create() 启动的线程相关，如果设置该位，相应的线程将不会被绑定。命令行中的核列表与 taskset 的使用方式相同。对于一个由 Intel 编译器生成的 OpenMP 程序的典型配置方式如下：

281

```
$ export OMP_NUM_THREADS=4
$ export KMP_AFFINITY=disabled
$ likwid-pin -s 0x1 -c 0,1,4,5 ./stream
[likwid-pin] Main PID -> core 0 - OK
-------------------------------------------
 Double precision appears to have 16 digits of accuracy
 Assuming 8 bytes per DOUBLE PRECISION word
-------------------------------------------
```

```
9  Array size =   20000000
10 Offset     =          32
11 The total memory requirement is  457 MB
12 You are running each test  10 times
13 --
14 The *best* time for each test is used
15 *EXCLUDING* the first and last iterations
16 [pthread wrapper] PIN_MASK: 0->1 1->4 2->5
17 [pthread wrapper] SKIP MASK: 0x1
18 [pthread wrapper 0] Notice: Using libpthread.so.0
19         threadid 1073809728 -> SKIP
20 [pthread wrapper 1] Notice: Using libpthread.so.0
21         threadid 1078008128 -> core 1 - OK
22 [pthread wrapper 2] Notice: Using libpthread.so.0
23         threadid 1082206528 -> core 4 - OK
24 [pthread wrapper 3] Notice: Using libpthread.so.0
25         threadid 1086404928 -> core 5 - OK
26 [... rest of STREAM output omitted ...]
```

这是著名的 STREAM 基准测试集在两个 Nehalem 处理器上运行 4 线程的输出结果，为了阻止程序使用 Intel 编译器默认配置亲缘性，必须在程序运行前（第 2 行）设置 KMP_AFFINITY 外壳变量。前缀为“[likwid-pin]”或“[pthread wrapper]”的行为 likwid-pin 的诊断输出。在启动其他线程之前，第 4 行绑定主线程到第一个核上。在第一个 OpenMP 并行区，重载的 pthread_create() 函数输出该线程所运行的核 ID（第 16 行）和掩码（第 17 行），但是第一个被启动的线程不应被绑定（这与 Intel 编译器相关），所以外部库跳过了该线程（第 19 行），其他线程都依据处理器列表而进行绑定。

在 MPI/OpenMP 混合程序中，MPI 库需要产生一些附加线程并且相应的掩码也需要改变。对 Intel MPI 和 Intel 编译器而言，运行混合程序代码时需要按下述方式使用 likwid-pin 工具：

```
1 $ export OMP_NUM_THREADS=8
2 $ export KMP_AFFINITY=disabled
3 $ mpirun -pernode likwid-pin -s 0x3 -c 0-7 ./a.out
```

每个节点启动一个 MPI 进程（由于有 -pernode 选项），每个进程有 8 个线程并被绑定到第 0 ～ 7 个核上，与简单的 taskset 工具不同，线程在启动后不允许在处理器组间迁移。不过，两种方式都有同样的缺点，即每个节点都只能使用单进程多线程的 MPI 进程方式。对于像每个插槽一个 MPI 进程的执行方式（见 11.3 节），还必须与 MPI 启动机制进行交互才能完成绑定过程。 282

A.2.2 程序控制亲缘性

如果不选择外部亲缘性控制工具，或者该类工具在系统中不可用，就需要在程序中显式进行绑定。每个操作系统都提供类似的系统调用或者库以完成类似功能。在 Linux 下，PLPA[T32] 是一个抽象 sched_setaffinity() 函数及相关系统调用的外部库。下面是一个 C 语言编写的 OpenMP 实例，每个线程都被绑定到通过线程标识索引的核标识上：

```
1 #include <plpa.h>
2 ...
3 int coremap[] = {0,4,1,5,2,6,3,7};
4 #pragma omp parallel
5 {
6   plpa_cpu_set_t mask;
7   PLPA_CPU_ZERO(&mask);
```

```
8    int id = coremap[omp_get_thread_num()];
9    PLPA_CPU_SET(id,&mask);
10   PLPA_NAME(sched_setaffinity)((pid_t)0, sizeof(mask), &mask);
11 }
```

mask 变量通过位掩码实现，通过设置某些位来标识某个线程可以在哪些处理核心上运行（这与 taskset 中的掩码使用方式一致）。通过 coremap[] 数组来建立一个特殊映射序列，在本例中，目标是将所有线程均匀地分布到前述 Nehalem 系统中的 8 个核上，因此已达到最优带宽利用率。在真实应用中，映射数组不应该被硬编码。

程序运行后，线程不允许迁移（可以被重新绑定）。也可以利用 PLPA 绑定 MPI 进程：

```
1 plpa_cpu_set_t mask;
2 PLPA_CPU_ZERO(&mask);
3 MPI_Comm_rank(MPI_COMM_WORLD,&rank);
4 int id = (rank % 4);
5 PLPA_CPU_SET(id,&mask);
6 PLPA_NAME(sched_setaffinity)((pid_t)0, sizeof(cpu_set_t), &mask);
```

这里考虑到简洁性没有使用映射数组。利用 PLPA 进行 MPI/OpenMP 程序绑定实例
283 如下：

```
1  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
2  #pragma omp parallel
3  {
4    plpa_cpu_set_t mask;
5    PLPA_CPU_ZERO(&mask);
6    int cpu = (rank % MPI_PROCESSES_PER_NODE)*omp_num_threads()
7              + omp_get_thread_num();
8    PLPA_CPU_SET(cpu,&mask);
9    PLPA_NAME(sched_setaffinity)((pid_t)0, sizeof(cpu_set_t), &mask);
10 }
```

上面我们用原始处理器整数标识，没有指定诸如 cache 组和插槽之类的组件（如果绑定机制支持这些组件）。在本书写作过程中，已经有大量的提供简单易用的亲缘性工具的研究成果，包括提供比 PLPA 更为全面的解决方案的“hwloc”工具包 [T33]。

注意 PLPA 和类似的接口只支持 C 语言，编写 Fortran 程序需要编写一个 C 语言的调用接口层。

A.3 非页面首次访问分配策略

页面首次访问分配策略适用于很多 ccNUMA 环境和操作系统，除了静态调度也没有其他更好的替换策略。尽管如此，通过 8.3.1 节和习题 8.1 的分析，由于影响程序执行性能的主要因素，即负载均衡的需要，我们也需要提供动态或者指导性调度。另一个类似的问题是当系统支持 OpenMP 任务时产生的 [O58]。应该充分将内存页面均匀分布（轮询方式）至整个局部区域，一边提高内存访问并行性。这也可以通过首次访问分配策略来实现，不过需要正确的初始化（例如标准循环）。如果初始化过程位于用户代码之外则会产生一些问题。

在 Linux 系统下，numactl 工具提供了非常方便的基于全局的页面分配策略，而且不需要改变应用程序代码。该工具还可提供其他功能，包括 SYSV 共享内存的放置策略以及类似 tastset 一样提供进程绑定的方法。这里我们忽略其他功能而只关心有关 NUMA 的功能：

```
1 numactl --hardware
```

可以利用 numactl 的输出检查系统局部性域中的可用内存，在 A.1 节提到的 ccNUMA
284 系统节点的输出如下：

```
$ numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 8 9 10 11
node 0 size: 6133 MB
node 0 free: 2162 MB
node 1 cpus: 4 5 6 7 12 13 14 15
node 1 size: 6144 MB
node 1 free: 5935 MB
```

在 numactl 中，节点表示一个局部性域。在 LD1 中有足够的内存可用，但是由于有程序运行或者内存被文件系统缓存所用（见 8.3.2 节），在 LD0 上只有 2G 内存。该工具还显示了每个核标识都属于哪一个节点。

针对页面分配策略有以下几种选项：

```
numactl [--interleave=nodes] [--preferred=node]
        [--membind=nodes] [--localalloc] <command> [args] ...
```

命令行中的选项 --interleave 设置内存页面的交替分配策略，与首次访问分配策略不同，页面将会以轮询方式在指定的 LD 列表中循环分配：

```
$ export OMP_NUM_THREADS=16          # using all HW threads
$ numactl --interleave=0,1 ./a.out   # --interleave=all for all nodes
```

如果在制定的 LD 上没有足够的可用内存，就会在其他 LD 上进行分配。交替分配策略可以用来快速检测具有 NUMA 局部性或者冲突特性的程序的执行结果：如果未改变的程序仅在运行时设置 numact1--interleave 模式下性能更高，用户需要检查数组初始化代码。

除了轮询分配策略，如果节点空闲内存足够的话，还可以进行节点级别（--prefered =node）页面分配，或者通过设置 --membind=nodes 在一组节点上进行分配。在后者的情况下，如果标识满配，程序会崩溃。选项 --localalloc 类似，不过总是选择映射当前节点，即在 numactl 初始启动节点上进行分配。

如果需要更为细粒度的页面分配方式，可以利用 Linux 系统中的 libnuma 库 [139]，它允许在调用内存分配后（例如通过 malloc）以及初始化前选择 NUMA 策略。

285 ~ 286

附录 B

Introduction to High Performance Computing for Scientists and Engineers

习题解答

习题 1.1

运行时间受除法和储存在寄存器中的数据主导，所以我们可以假设每次循环迭代的周期数等于除法吞吐量（这里假设等于延迟）。当设计 SIMD 操作时，需要注意；如果 p 个除法能够并行执行，那么基准测试需要将估计的除法延迟除以 p。

目前 x86 架构处理器双精度的除法延迟在 20 ～ 30 周期之间。

习题 1.2

正如 1.2.3 节中解释的，在迭代 i 中 ofs=1 阻塞流水线直到 $i-1$ 次迭代完成。阻塞时间接近于流水线深度。如果 ofs 增加，阻塞时间将越来越小直到最后，如果 ofs 比流水线深度大，那么阻塞将消失。注意，我们忽略编译器使用 SIMD 指令向量化代码所带来的递归影响。

习题 1.3

无论什么时候从内存中预取的数据流远远小于一个内存页，预取器倾向于读取比需要多更多的数据 [O62]。如果太多预取请求排队，那么这种影响可以通过硬件取消预取流来改善——但是不能消除。

注意如果流很短，则 TLB 失效将是另一个需要考虑的重要因素。详细介绍请参考 3.4 节。

习题 1.4

（a）没有预取，获取两个 cache 行所需时间是延迟的两倍（100ns）加上纯数据传输（带宽）两倍（10ns），总共 220ns。既然一个 cache 行有 4 项，则在这些数据上可以执行 8 个 Flop（每项两个乘法和两个加法）。因此，预期性能是 8 Flop/220 ns = 36 MFlop/s。

（b）根据式（1-6），需要 1+100/10=11 预取来隐藏延迟。注意这个结果不依赖并发流的数量仅当我们假设可获得的带宽独立于这个数。

（c）如果 cache 行长度增加，则延迟不变但是将花费更多时间传输数据，也就是，整体传输时间增加。当 cache 行是 64 字节时，我们需要 1+100/20 = 6 个预取，并且当 128 字节时仅需要 1+100/40 ≈ 4。

（d）如果没有延迟，传输两个 cache 行花费 20ns，并且在这段时间 8 个 flop 可以被执行。这将带来理论性能 4 × 108 Flop/s, 或者 400 MFlop/s。

习题 2.1

由于依赖于数据是否从内存中被预取，条件分支带来的影响是巨大的。对于 cache 外的数据，也就是，N 很大时，代码执行和标准向量三元组一样，而和 C 内容无关。然而当 N 很小时，如果分支不能预测的话，性能下降很大，即对于 C 值的随机分布来说。如果 C（i）总是小于或者大于 0，则性能可以恢复，这是因为大多数情况下分支是可以被预测的。

注意编译器能够做有趣的事情，例如对于循环，尤其涉及 SIMD 操作循环。如果执行实际基准测试，则尽量取消编译器的 SIMD 功能以便获取一个更清晰的情况。

习题 2.2

在“SIMD 单元”中的操作必须独立，但是它们可能依赖于距离很远的数据，不管是负还是正。尽管对于 offset<0，流水线次优的，但是 4 倍数的偏移（正或负）没有阻止 SIMD 向量化。注意，如果在编译时偏移不确定的话，那么编译器总是禁止在这样的循环中 SIMD 向量化。你能想象 SIMD 向量化这个代码，即使 offset 不是 4 的倍数吗？

习题 2.3

函数中或者块中的一个 C 风格数组被分配在栈上。这是一个基本上没有开销的操作，所以它将不会对性能有什么影响。但是，依据栈大小限制，这个选项不总是可能。

习题 2.4

STL 的 std::vector<> 类有容量和大小概念的区别。如果这里有已知的向量长度的上界，那么赋值可能不需要重新分配空间：

```
1  const int max_length=1000;                                          288
2
3  void f(double threshold, int length) {
4    static std::vector<double> v(max_length);
5    if(rand() > threshold*RAND_MAX) {
6      v = obtain_data(length);  // no re-alloc required
7      std::sort(v.begin(), v.end());
8      process_data(v);
9    }
10 }
```

习题 3.1

如果 s 比 DP 字中的 cache 行长度小，连续的 cache 行仍然从内存中被读取。假设预取机制能够隐藏所有延迟，则内存接口饱和但是我们仅使用传输数据的 1/s。因此，应用程序可获取带宽（实际加载和存储）将降到 1/s，向量三元组性能也如此。对于大于 cache 行的 s，性能保持不变，这是因为每个 cache 行每项被准确使用，而不管 s 有多大。当然，预取将在某些点终止，并且性能将下降并最终由延迟决定（见习题 1.4）。

如果使用非临时存储写 A()，那么这些考虑的情形会改变吗？

习题 3.2

正如 3.1.2 节中所示，如果片上第二个核空闲，那么单个 Intel Xeon 5160 核拥有 12 GFlop/s 的峰值性能和 $B_m^X = 0.111$ W/F 理论机器均衡。STREAM TRIAD 基准测试产生的有效均衡仅是这里的 36%。对于向量 CPU，峰值性能是 16 GFlop/s。理论和高效的机器均衡相同：$B_m^V = 0.5$ W/F。这里考虑的所有 4 个内核都是只读的（因为内存循环指示变量是 i，所以（a）内核中 Y(j) 可以保存在寄存器中），所以写分配传输不是问题。虽然不存在只读的 STREAM 基准测试，但是 2：1 加载和存储比的 TRIAD 已经足够接近，特别是 TRIAD 之间的有效带宽没有太多差别时，比如，Xeon 上的 COPY。

我们用 P_X 和 P_V 分别表示 Xeon 和向量 CPU 上的期望性能：

（a）$B_c = 1$ W/F

$P_X = 12$ GFlop/s $\cdot$ 0.36 $\cdot B_m^X / B_c = 400$ MFlop/s

$P_V = 16$ GFlop/s $\cdot B_m^V / B_c = 8$ GFlop/s

（b）$B_c = 0.5$ W/F。期望性能两倍于（a）。

（c）和（a）相等。

（d）仅算上代码显式加载时，代码均衡似乎是 $B_c = 1.25$ W/F。但是，有效值依赖于数组

289 K() 的内容：当 K(i) 中的项连续时，所给的值才是正确的。如果，比如，K(i) = 常数，只有 B() 项被加载，所以在 cache 依赖的 CPU 上有 $B_c^{min,X}$ = 0.75 W/F。一个向量 CPU 没有 cache 优势，所以它必须反复加载同样值，所以代码均衡不变。另一方面，它可以高效地分散加载，所以 B_c^V = 1.25 W/F 独立于 K() 内容（实际上，和理想有偏差，例如，聚集操作比连续流效率差）。因此，向量处理器应该是 $P_V = 16\ \text{GFlop/s} \cdot B_m^V / B_c^V = 6.4\ \text{GFlop/s}$。

基于 cache CPU 的最差情况是有大的相关区域完全随机的 K(i)，所以每次从中加载都会引起 cache 行全部读，而这时在无效前仅有单独一项被使用。如果 cache 行是 64 字节大小，那么 $B_c^{max,X}$ = 4.75 W/F。因此，三种情形如下：

① K(i) = 常数：

$B_c^{min,X}$ = 0.75 W/F，和

$P_X = 12\ \text{GFlop/s} \cdot 0.36 \cdot B_m^X / B_c^{min,X} = 639\ \text{MFlop/s}$

② K(i) = i:

B_c = 1.25 W/F，和

$P_X = 12\ \text{GFlop/s} \cdot 0.36 \cdot B_m^X / B_c^X = 384\ \text{MFlop/s}$

③ K(i) = 随机值：

B_c = 4.75 W/F，和

$P_X = 12\ \text{GFlop/s} \cdot 0.36 \cdot B_m^X / B_c^{max,X} = 101\ \text{MFlop/s}$

这个估计值仅仅是一个上界，这是因为预取并没有工作。

注意一些 Intel x86 处理器每次失效时总是预取两个连续 cache 行（你能想象为什么吗？），这导致在最差情形下代码均衡的增加。

习题 3.3

我们将初始运行时间归一化为 $T = T_m + T_c = 1$，其中 T_m = 0.4 和 T_c = 0.6。应用程序的这两个部分有不同特点。0.04 W/F 的代码均衡意味着性能不被内存带宽限制，而是 CPU 核中的另一个因素。如果峰值性能达到双倍，那么这部分的绝对运行时间很可能减少一半，这是因为 0.06 W/F 的机器均衡仍然比代码均衡大很多。应用程序另一部分运行时间将不会变化，因为内存受限是 0.5 W/F 的代码均衡。总之，整体运行时间将是 $T = T_m + T_c/2 = 0.7$。

如果 SIMD 向量长度（也就是峰值性能）增长，机器均衡将减少很多。在机器均衡为 0.04 W/F 的情况下，应用程序先前 CPU 约束的部分将变成内存约束的。从这个观点，峰值性能提高并不能改善它的运行时间。因此整体时间是 $T_{min} = T_m + T_c/3 = 0.6$。

290 考虑 Amdahl 法则并且上面的考虑的确是它背后概念的利用。但是，这里有点复杂，这是因为在某些关键机器均衡中，应用程序的 CPU 约束部分变成内存约束以至于到达绝对性能极限，而与 Amdahl 法则相反的它是一个逼近值。

代码清单 B-1 用三路循环块实现的三维 Jacobi 算法实现

```
1  double precision :: oos
2  double precision, dimension(0:imax+1,0:jmax+1,0:kmax+1,0:1) :: phi
3  integer :: t0,t1
4  t0 = 0 ; t1 = 1 ; oos = 1.d0/6.d0
5  ...
6  ! loop over sweeps
7  do s=1,ITER
8    ! loop nest over blocks
9    do ks=1,kmax,bz
10     do js=1,jmax,by
```

```
11        do is=1,imax,bx
12          ! sweep one block
13          do k=ks,min(kmax,ks+bz-1)
14            do j=js,min(jmax,js+by-1)
15              do i=is,min(imax,is+bx-1)
16                phi(i,j,k,t1) = oos * ( &
17                  phi(i-1,j,k,t0)+phi(i+1,j,k,t0)+ &
18                  phi(i,j-1,k,t0)+phi(i,j+1,k,t0)+ &
19                  phi(i,j,k-1,t0)+phi(i,j,k+1,t0) )
20              enddo
21            enddo
22          enddo
23        enddo
24      enddo
25    enddo
26    i=t0 ; t0=t1; t1=i   ! swap arrays
27  enddo
```

习题 3.4

相对于二维 Jacobi，我们预期当问题规模增大时，性能会大幅度下降：如果两个连续 k 平面能够放在 cache 中，那么 6 次加载只有 1 次引起 cache 失效。如果 cache 足够大而能放下 k 平面中连续两个 j 行，那么将有 3 个加载需要访问内存（涉及模板更新的每个 k 平面有一次）。最后，当问题规模很大后，只有单个加载能够放在 cache 中。当问题规模非常大，或者计算区域有一个椭圆形状时，将会发生这种情况，当然实际情形并不常见。

正如矩阵转置例子所示，循环分块可以消除性能下降。整体的一次扫描可以分成 bx × by × bz 大小的子块多次扫描。可以选择足够小的字块使得连续两个 k 层能够放在外层 cache 中。一个可能的实现如代码清单 B-1 所示。注意，我们并不考虑数据对齐或者非临时存储优化。 291

上面所示的循环分块是所谓的空间分块（spatial blocking）。如果问题规模对于外层 cache 很大，那么所有性能下降都可以消除，也就是，它能保持 0.5 W/F（每 6 个 Flop 3 个字，包括写分配）的代码均衡。问题仍然存在：块大小的最优选择是多少？原则上，在 i 和 j 方向上块应该足够小使得最少两个连续 k 平面能够放在外层 cache 中以便被核访问。应该考虑“安全因素”，这是因为全部 cache 永远不会使用完。注意，硬件预取器（尤其在 x86 处理器上）和 TLB 失效所带来的开销需要内层循环中处理的内存流应该不比一个页面小太多。

为了减少代码均衡，不得不在分块方案中包含迭代循环，当每项都被加载 cache 中时，更新多个模板。这是所谓的时间分块（temporal blocking）。虽然概念很直接，但是它的优化实现、性能特性，以及多核结构交互仍是活跃的研究领域 [O61，O73，O74，O75，O52，O53]。

习题 3.5

模板代码能从内层循环展开获益，这是因为可以减少从 cache 到寄存器中的加载。为了便于展示，我们考虑一个简单“两点模板”：

```
1 do i=1,n-1
2   b(i) = a(i) + s*a(i+1)
3 enddo
```

在每次迭代中，当更新 b（i）时，需要两次加载和一次存储。但是，a（i+1）能够保存在寄存器中并且在下次迭代时能够立即被重用（在这里我们忽略可能的剩余循环）：

```
do i=1,n-1,2
  b(i) = a(i) + s*a(i+1)
  b(i+1) = a(i+1) + s*a(i+2)
enddo
```

这将从 a() 数组中的加载省去一半。但既然这个版本中没有使用循环展开，额外加载总是来自于 L1 cache，所以这是一个 cache 内优化。对于较长的、内存约束循环来说，循环展开没有任何优势。在简单例子中，编译器将会自动地对内层循环的变体采取展开。

留给读者弄清 Jacobi 模板中如何应用内层循环展开。

习题 3.6

原则上，如果循环嵌套是“矩形的”，也就是，
292 如果内层循环边界不依赖于外层循环指示变量，那么循环展开和阻塞是仅有的可能性。这个条件在这里不成立，但是可以将矩形块组从三角矩阵中分离而对剩下类循环剥离单独处理。图 B-1 显示它是怎么工作的。对于四路展开和阻塞来说，$4 \times r\ (r \geqslant 1)$ 大小的块通过展开嵌套循环被遍历（阴影部分）。剩下的部分采用完全循环展开：

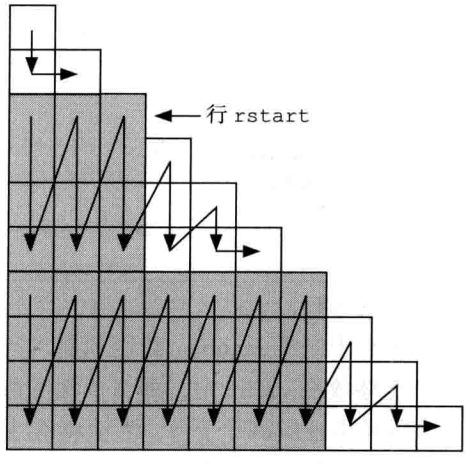


图 B-1 三角矩阵和向量乘法的四路循环展开和阻塞。箭头显示了如何遍历矩阵元素。阴影块是矩形的，因此是展开和阻塞的候选。白色矩阵元素必须单独处理

```
rstart = MOD(N,4)+1

do c=1,rstart-1 ! first peeled-off triangle
  do r=1,rstart-c
    y(r) = y(r) + a(c,r) * x(c)
  enddo
enddo

do b = rstart,N,4 ! row of block start
  ! unrolled loop nest
  do c = 1,b
    y(b) = y(b) + a(c,b) * x(c)
    y(b+1) = y(b+1) + a(c,b+1) * x(c)
    y(b+2) = y(b+2) + a(c,b+2) * x(c)
    y(b+3) = y(b+3) + a(c,b+3) * x(c)
  enddo

  ! remaining 6 iterations (fully unrolled)
  y(b+1) = y(b+1) + a(b+1,b+1) * x(b+1)
  y(b+2) = y(b+2) + a(b+1,b+2) * x(b+1)
  y(b+3) = y(b+3) + a(b+1,b+3) * x(b+1)
  y(b+2) = y(b+2) + a(b+2,b+2) * x(b+2)
  y(b+3) = y(b+3) + a(b+2,b+3) * x(b+2)
  y(b+3) = y(b+3) + a(b+3,b+3) * x(b+3)
enddo
```

293

习题 3.7

这里代码有如下可能性能问题。

- 内存循环由计算开销很大的三角函数主导。
- 这里有开销相当大的整数取模运算（要比其他整数的算术或者逻辑预算慢许多）。
- 跨步访问矩阵 mat 和 s，是因为内层循环随着 j.SIMD 向量化而无法采用。

将开销大的运算从内存循环提取出来是首要而简单的步骤（尽管这实际上由编译器完成）。同时，我们应该使用位掩码来替代取模操作并对复杂三角函数采用更简单的计算替代：

```
do i=1,N
  val = DBLE(IAND(v(i),255))
  val = -0.5d0*COS(2.d0*val)
  do j=1,N
    mat(i,j) = s(i,j) * val
  enddo
enddo
```

尽管这种优化提升性能不多，但是在内层循环中的跨步访问是有风险的，尤其当N很大时（参考3.4节）。我们不能简单交换循环层次，因为这样会导致将开销大的操作又移动到内存循环。注意余弦仅在256个不同值上计算，所以可以构造一张表格：

```
double precision, dimension(0:255), save :: vtab
logical, save :: flag = .TRUE.
if(flag) then    ! do this only once
  flag = .FALSE.
  do i=0,255
    vtab(i) = -0.5d0*COS(2.d0*DBLE(i))
  enddo
endif
do j=1,N
  do i=1,N
    mat(i,j) = s(i,j) * vtab(IAND(v(i),255))
  enddo
enddo
```

在内存约束情形中，也就是，当N很大时，由于间接操作开销很小，并且vtab（）将永久缓存，所以这是一个好的解决方案。更好的是，表格仅被计算一次。但是，如果问题规模刚好能够放进cache中，那么SIMD向量化就变得很重要。一种向量化内存循环的方式是不仅构建256个三角函数值而且构建在s(i,j)之后的整个函数值：

294

```
double precision, dimension(0:255), save :: vtab
double precision, dimension(N) :: ftab
logical, save :: flag = .TRUE.
if(flag) then    ! do this only once
  flag = .FALSE.
  do i=0,255
    vtab(i) = -0.5d0*COS(2.d0*DBLE(i))
  enddo
endif
do i=1,N         ! do this on every call
  ftab(i) = vtab(IAND(v(i),255))
enddo
do j=1,N
  do i=1,N
    mat(i,j) = s(i,j) * ftab(i)
  enddo
enddo
```

因为在函数调用之间v()可能变化，所以ftab()每次都必须重新计算，但是内层循环现在SIMD向量化了。

习题3.8

TLB“cache行”是内存页，所以失效开销必须和将页加载到cache中所花费的时间对比。现在，内存带宽是每个核几个GB/s，所以导致传输4kB的页大概需要1μs。不同体系结构中TLB失效开销是不同的，但是它通常要小于100个CPU周期。在时钟频率大概

2GHz 时，纯流化代码上的每页 TLB 失效对性能没有太大影响。如果在下次之前传输远小于一个页，这当然将改变。

一些系统有更大内存页，或者可以配置使用。最好是一个应用程序的整个工作集能够映射到 TLB 中使得失效数目为零。即使不是这样的情形，大页面也仍能够提高 TLB 的命中率，这是因为在任何给定时间一个新页面被命中的概率很小。但是，采用大页面会减少可获取的 TLB 项的数量。

习题 4.1

静态路由为每对通信伙伴分配固定的网络路线。由于 2：3 的超额认购，一些连接将会获取较少的到主干上的带宽，因为在主干连接上均匀分发叶子是不可能的（在叶子转换处的叶子数不是它与主干连接数的倍数）。因此，网络变得更加不平衡，超出了由静态路由及超
295 额认购单独带来的竞争效应。

习题 5.1

在计算后面隐藏通信是一件听起来简单但通常在实际中不易实现的一件事情，详见 10.4.3 节及 11.1.3 节。无论如何，假定是可能的，位于强扩展加速比函数（见式（5-30））分母上的并行执行时间变为

$$\max\left[(1-s)/N,(\kappa N^{-\beta}+\lambda)\mu^{-1}\right] \tag{B-1}$$

这描绘了从完美到部分隐藏通信的一个分界点，当

$$\mu(1-s)=\kappa N_c^{1-\beta}+\lambda N_c \tag{B-2}$$

由于方程的右端函数在 $0\leqslant\beta\leqslant 1$ 时关于 N_c 是严格单调的，因此在速度慢的计算机上（$\mu>1$）分界点总是会转换为更大的 N。尽管 N_c 精确地（至少因子 μ 是合适的）被比率 $N_c(\mu)/N_c(\mu=1)$ 所给出，但是求解 N_c 并不容易。幸运的是，研究重要限制 $\lambda=0$ 以及 $\kappa=0$ 的情况已经足够了。为了消除延迟，

$$N_c(\lambda=0)=\left(\frac{\mu(1-s)}{\kappa}\right)^{1/(1-\beta)} \tag{B-3}$$

以及

$$\frac{N_c(\mu>1)}{N_c(\mu=1)}\bigg|_{\lambda=0}=\mu^{1/(1-\beta)}>\mu \tag{B-4}$$

在延迟主导的限制 $\kappa=0$ 时，我们可以立即得到

$$\frac{N_c(\mu>1)}{N_c(\mu=1)}\bigg|_{\kappa=0}=\mu$$

至此，我们已经推导出了运行慢的计算机的一个额外收益：如果通信能在计算后面被隐藏，它将明显地变成一个确定的 N，这会比标准计算机至少大 μ 倍。

对于弱扩展，你能做相同的分析吗？

习题 5.2

对于强扩展及延迟主导的通信，执行的经过时间（“求解时间”）为

$$T_w=s+\frac{1-s}{N}+\lambda \tag{B-6}$$

其中 s 为不可并行部分，λ 为延迟。故 N 个处理器的花销为 NT_w，产生的花销 - 经过时间之积为

296

$$V=NT_w^2=N\left(s+\frac{1-s}{N}+\lambda\right)^2 \tag{B-7}$$

当

$$\frac{\partial V}{\partial N}=\frac{1}{N^2}[(1+\lambda N+(N-1)s)(s-1+N(\lambda+s))]=0 \tag{B-8}$$

时取得极值。这个方程关于 N 唯一的正数解为

$$N_{\text{opt}}=\frac{1-s}{\lambda+s}。 \tag{B-9}$$

有趣的是，如果我们假设为带有晕环交换的二维区域分解，那么

$$T_{\text{w}}=s+\frac{1-s}{N}+\lambda+\kappa N^{-1/2} \tag{B-10}$$

结果与式（B-9）相同，也就是与 κ 无关。然而，这是一个特例；通信开销中 N 的任何其他幂次会导致本质上的不同结果（以及一个更复杂的导数）。

最后，在 $N=N_{\text{opt}}$ 时获得的加速比为

$$S_{\text{opt}}=\left(2(s+\lambda)+\kappa\sqrt{\frac{\lambda+s}{1-s}}\right)^{-1} \tag{B-11}$$

用最大加速比 $S_{\max}=1/(\lambda+s)$ 与其比较，得到

$$\frac{S_{\max}}{S_{\text{opt}}}=2+\frac{\kappa}{\sqrt{(1-s)(\lambda+s)}} \tag{B-12}$$

因此当 $\kappa=0$ 时，“最佳点”为最大加速比的一半；对于有限大小的 κ 会更低。

当然，不管并行应用程序的通信需求是什么，如果工作负载对于单一串行应用程序的运行是“可容忍的”，并且必须执行很多这样的运行，那么使用一个并行计算机的最有效方式是吞吐量模式。在吞吐量模式中，同一串行代码的很多实例同时运行在一个资源管理系统的控制下（在所有的产品机上都是这样的）。这样使得所有的资源得到最好的利用。显然，这种方式的工作负载对于带有昂贵互连网络的大规模并行计算机是不适合的。

习题 5.3

对于强扩展，同步和通信开销一样会引起最后的减速。对于弱扩展，线性的可扩展性遭到破坏；线性同步开销甚至会造成饱和。

习题 5.4

忽略通信方面（如将数据移入和移出加速器的开销），我们可以通过假设应用程序的加速部分执行得比主机部分快 α 倍而其余部分保持不变，来对这个情形建模。使用 Amdahl 定律，s 为主机部分，$p=1-s$ 为加速的部分。因此，渐进的性能由主机部分主导；例如，当 $\alpha=100$ 且 $s=10^{-2}$ 时，加速比仅为 50，这意味着我们浪费了加速器一半的计算能力。为了在 $0<r<1$ 时得到 $r\alpha$ 倍的加速比，我们需要求解 297

$$\frac{1}{s+\frac{1-s}{\alpha}}=r\alpha \tag{B-13}$$

导出

$$s=\frac{r^{-1}-1}{\alpha-1} \tag{B-14}$$

当 $r = 0.9$，$\alpha = 100$ 时，得到 $s \approx 1.1 \times 10^{-3}$。这个结论表明，高效使用加速设备需要原始执行时间的主要部分（超过 $1-11\alpha$）被移至专门的硬件上。Amdahl 在他最原始的论文中，沿着在 ILLIAC IV 超级计算机 [R40] 上对比“加速执行”和“管家及数据管理”方面的努力方向顺便得到了他的著名定律，它实现了一个大规模数据并行的 SIMD 编程模型：

在这一点上可以得出的一个十分明显的结论是在获得高并行处理速率上去努力是浪费时间，除非它伴随着近乎相同规模的串行处理速率 [M45]。

这一陈述与上面描述的情形完美符合。实质上，它在 5.3.5 节中已经用数学方法推导出来，尽管上下文稍有不同。

有人可能认为扩大（加速的）问题规模会缓解这一问题，但这一观点由于加速器内存大小的限制而遭到质疑。为了保持串行（或非加速的）部分和通信开销可控，计算单元（核、插槽、节点、加速器）的性能越大，相应的内存就会越大。

习题 6.1

由于 noise 变量在声明时初始化，所以在子过程 f() 中它含有隐式 SAVE 属性。因此初始值仅在首次调用时被设置，这正是串行代码所想要的。但是，在并行区域中调用 f() 使得 noise 成为共享变量，并且存在竞争条件。要想修改这问题，要么将 noise 作为 f() 的参数（类似于线程安全随机数生成器的种子），要么它的更新应该由同步结构保护。

习题 6.2

298 关键部分是线程安全的随机数生成器。根据 OpenMP 标准 [P11]，在 Fortran 90 中的 RANDOM_NUMBER() 原语子过程应该是线程安全的，所以在这里可以使用。但是，会有性能影响（为什么？）并且通常最好避免使用内建生成器（详尽讨论请参考 [N51]），所以我们假设这里存在和 POSIX 中 rand_r() 函数相同的 ran_gen() 函数：使用整数随机种子，这个种子每次调用都被更新并且被存放在每个线程各自的调用函数中。函数返回值在 0 ～ 2^{31} 之间，很容易被转换成所需范围的浮点数：

```
1   integer(kind=8) :: sum
2   integer, parameter :: ITER = 1000000000
3   integer :: seed, i
4   double precision, parameter :: rcp = 1.d0/2**31
5   double precision :: x,y,pi
6  !$OMP PARALLEL PRIVATE(x,y,seed) REDUCTION(+:sum)
7   seed = omp_get_thread_num()    ! everyone gets their own seed
8  !$OMP DO
9   do i=1,ITER
10    x = ran_gen(seed)*rcp
11    y = ran_gen(seed)*rcp
12    if (x*x + y*y .le. 1.d0) sum=sum+1
13  enddo
14 !$OMP END DO
15 !$OMP END PARALLEL
16  pi = (4.d0 * sum) / ITER
```

在第 7 行中，线程私有的种子设置为不同值。在第一象限中的 1/4 圆“命中次数”是通过 sum 中的归约操作计算的（在所有线程上求和），并且使用第 16 行计算最终结果 π。

为每个线程使用不同随机种子很重要，这是因为每个线程都产生伪随机数，那么随机误差和一个线程运行的结果一样。

习题 6.3

func() 的计算并不是一定需要临界区域保护（和 sum 计算刚好相反）。它可以在外面计

算，这样两个临界区域就不互相干涉了：

```
1  !$OMP PARALLEL DO PRIVATE(x)
2    do i=1,N
3      x = SIN(2*PI*DBLE(i)/N)
4      x = func(x)
5  !$OMP CRITICAL
6      sum = sum + x
7  !$OMP END CRITICAL
8    enddo
9  !$OMP END PARALLEL DO
10   ...
11   double precision FUNCTION func(v)
12   double precision :: v
13 !$OMP CRITICAL
14   func = v + random_func()
15 !$OMP END CRITICAL
16   END SUBROUTINE func
```

299

习题 6.4

组中的所有线程都应该到达同步点。但是在工作共享组中是无法保证的。

习题 6.5

感谢来自 Red Hat 的 Jakub Jelinek 提供的这个答案。如果我们意识到 opt(n) = up**n，那么可以立即并行循环，但是指数计算开销很大，所以我们将不采取这种方法（尽管我们采取并行代码的单线程性能作为基线，扩展性会很好 [S7]）。如果我们确保这个线程之前计算的迭代是在 $n-1$（第 12 行），那么我们将采取 opt(n) 的初始（快速）版本。另一方面，如果我们在一个新的块组起始，那么指示变量将被跳过并且 opt(n) 只能以指数方式计算（第 14 行）：

```
1   double precision, parameter :: up = 1.00001d0
2   double precision :: Sn, origSn
3   double precision, dimension(0:len) :: opt
4   integer :: n, lastn
5
6   origSn = 1.d0
7   lastn = -2
8
9  !$OMP PARALLEL DO FIRSTPRIVATE(lastn) LASTPRIVATE(Sn)
10   do n = 0,len
11     if(lastn .eq. n-1) then        ! still in same chunk?
12       Sn = Sn * up                 ! yes: fast version
13     else
14       Sn = origSn * up**n          ! no: slow version
15     endif
16     opt(n) = Sn
17     lastn = n                      ! storing index
18   enddo
19 !$OMP END PARALLEL DO
20   Sn = Sn * up
```

LASTPRIVATE(Sn) 语句确保 Sn 在循环结束后和串行情形有相同值。当并行区域开始时，FIRSTPRIVATE(lastn) 为 lastn 赋于初值给私有副本。这纯粹为了方便，因为我们完全可以通过拆开组合 PARALLEL DO 指令来手动复制。

虽然这个解决方案使用所有 OpenMP 循环调度选项，但是它对于块组大小很小的静态或者动态调度尤其慢。在“STATIC,1”的特殊例子中，和开始使用的指数方法一样慢。

300

习题 6.6

这种优化又被称为扭曲循环（Loop skewing）。从一个特殊位置开始，所有能被同时更新的位置都是所谓的超平面的一部分并且满足条件：$i + j + k = \text{const.} = 3n$，这里 $n \geqslant 1$。当

所有内层循环迭代独立时，流水化执行成为可能，使得向量流水线得到了高效使用（请参考6.1 节）。通过超平面方法，基于 cache 的处理器将遭受不稳定的访问模式。一个确定的模板更新所取得的 cache 行在 cache 中待的时间不够充分利用空间局部性。带宽利用（正如应用程序所示）因此很差。这是为什么嵌套波前并行化 Gauss-Seidel 循环的标准形式更适合于基于 cache 的微处理器，这里忽略了不充分流水线所带来的性能损失。

习题 7.1

在 C/C++ 中，归约语句并不能应用于数组。因此归约必须手动完成。firstprivate 语句有助于初始化 s[] 的私有副本；在一个“真”的 OpenMP 归约中，这同样自动完成：

```
1   int s[8] = {0};
2   int *ps = s;
3
4  #pragma omp parallel firstprivate(s)
5  {
6  #pragma omp for
7    for(int i=0; i<N; ++i)
8      s[a[i]]++;
9  #ifdef _OPENMP
10 #pragma omp critical
11   {
12     for(int i=0; i<8; ++i) {  // reduction loop
13       ps[i] += s[i];
14     }
15   } // end critical
16 #endif
17 } // end parallel
```

使用条件编译，如果不使用 OpenMP 编译，我们将跳过显式归约。

如果归约操作符不被 OpenMP 直接支持，那么在 Fortran 中需要采取同样相似的手段。同样，重载 C++ 操作符在归约语句是不被允许的，甚至对于标量类型也是如此。

习题 7.2

图 B-2 显示了当问题规模大约为 600^2 时的情形：单个 4MB 的 L2 cache 太小而不能容纳
301 工作集，但是 8MB 足够。从一（或者两）个线程到四个线程的加速比是 5.4。

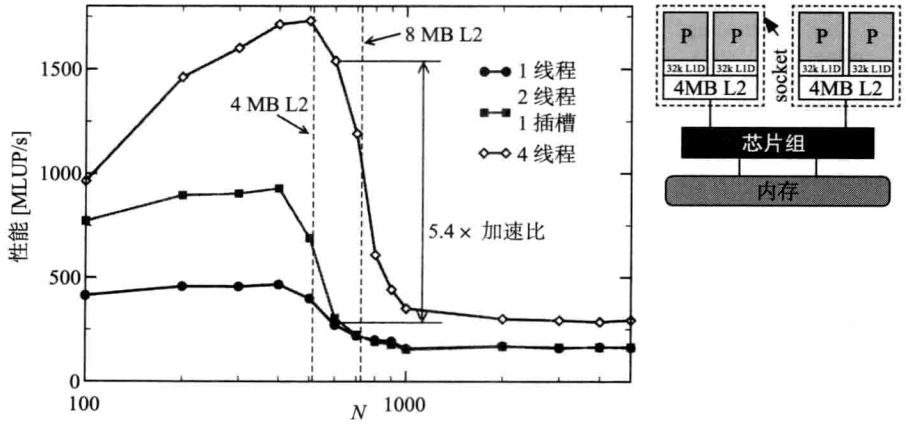


图 B-2 二维 Jacobi 解法器的超线性加速比（和图 6-3 相同的数据）。当问题规模为 600^2 时，工作集太大而不能容纳单个槽的 L2 cache 中，所以性能接近于内存约束（实心符号）情形。使用额外两个线程（空心方块），可以获得 8MB cache，这将大到足以容纳所有数据。因此获得 5.4× 的加速比

当工作集能够放入 cache 中时，网格更新如此快以至于，尤其是当线程数据很大时，典型 OpenMP 开销尤其是隐式栅栏（请参考 7.2.2 节）可能主导程序运行。要想找到这种标准，我们必须和使用传统同步开销的扫描时间比较。观察 cache 内双线程最大性能（实心方块），我们估计单遍扫描花费 170μs。作为一个粗略的指导，如果栅栏是高效实现的，那么我们假设在共享内存机器同步所有线程需要大概每个槽 1μs[M41]。因此，在标准双或者四槽节点上，应该能够观察到这种问题的超线性加速。

习题 7.3

如果节点不使用 OpenMP 编译，那么它仍然产生正确结果，但是如果 R 没有初始化，结果则出错。如果我们使用 C(j) 作为归约变量，那么不但避免这种情况，而且在归约语句中仅允许命名变量。

习题 7.4

并行会带来开销，所以程序员使用的线程越少越好。最优数量是通过主存带宽的扩展性和线程数量相比给出的。如果六个线程中两个已经充分使用一个插槽的内存总线，那么运行多余四个线程没有什么意义。所有关于 cache 组和系统架构的细节没有太多相关。

注意共享 cache 并不一定能够提供最优带宽的扩展，所以这个原因也适用于 cache 内计算 [M41]。 302

习题 8.1

通常来说，性能（P）是工作量（W）除以时间（t）。我们选择 W=2，意味着两个内存页被分配到两个运行线程。如果没有非局部访问，每个块使用 t=1 的时间局部地执行，以便 $P = 2p$。一般，四个实例必须区分开来：

1）t=1 时，两个线程局部地访问页面。

2）两个线程不得不远程访问它们的页面：因为假设 LD 间的网络无限快速，所以每个内存总线都没有竞争，这时 t=1。

3）两个线程都在 LD0 访问它们的页面：由于它们内存总线存在竞争导致 t=2。

4）两个线程都在 LD1 访问它们的页面：由于它们内存总线存在竞争导致 t=2。

这四种情形等概率发生，所以流化两个页面的平均时间 $t_{avg} = (1+1+2+2)/4 = 1.5$。因此，$P = W/t_{avg} = 4p/3$，也就是，该代码比最佳局部访问慢 33%。

这个导数很简单，因为我们仅涉及两个局部区域，并且无论它们被映射到什么地方，基本的工作包都是两个页面。你能够一般化任意数量的局部区域分析过程吗？

习题 8.2

当块组很小时，有两个可能的理由不利于性能：

- 基于硬件预取机制的 x86 架构处理器实际上会适得其反。一旦预处理器被一定数量的连续的 cache 失效（这里是 2）激活后，它开始传输数据到 cache 直到当前页面末尾（或者直到取消）。如果块组大小小于一个页面，那么获取的一些数据将不需要并且这会浪费带宽和 cache 容量。块组大小越接近页面大小，影响越小。
- 一个小块组也将增加 TLB 失效的数量。TLB 失效对性能的影响强依赖于处理器本身。请参考习题 3.8。

习题 8.3

如果需要多流，那么相对于首次访问策略，循环布局策略可能产生好的单槽性能，这是因为部分带宽可以通过远程访问来满足。但是，这种策略带来的好处很大程度上依赖于硬件。 303

习题 8.4

并行内层归约循环很好，但是仅对于大规模问题适用，这是因为 OpenMP 开销和 NUMA 布局策略问题。但是，外层循环对不同迭代有不同负载，所以存在严重负载均衡问题：

```
!$OMP PARALLEL DO SCHEDULE(RUNTIME)
do r=1,N              ! parallel initialization
  y(r) = 0.d0
  x(r) = 0.d0
  do c=1,r
    a(c,r) =0.d0
  enddo
enddo
!$OMP END PARALLEL DO
...
!$OMP PARALLEL DO SCHEDULE(RUNTIME)
do r=1,N              ! actual triangular MVM loop
  do c=1,r
    y(r) = y(r) + a(c,r) * x(c)
  enddo
enddo
!$OMP END PARALLEL DO
```

为了得到好的布局，我们在 ccNUMA 系统中增加了并行初始化循环。

标准静态调度在这里当然不是一个好的选择。具有合适大小块组的指导性或者动态调度能够使负载均衡而不会带来太多开销，但是在 ccNUMA 上将导致非局部访问这是因为块组是在运行时动态分配。因此，仅有的合理选择是静态调度，并且块组大小使得矩阵行不用太靠近三角矩阵 tip。

那么 x() 的布局问题呢?

习题 8.5

并行循环调度是一个关键点。初始化和工作共享的循环调度应该是相同的。如果我们使用单循环，一个可能块组的大小将是循环迭代次数的倍数，将存放在 char 中。处理 D 类型对象的工作共享循环上的块组大小将参考 sizeof（D）的大小，并且 NUMA 布局将是错误的。

习题 9.1

交换发送和接收的顺序也适用于进程个数为奇数时。“剩下”的进程将不得不等待直到
304 其相邻进程完成通信。参见 10.3.1 节中有关“开链条”的讨论。

习题 9.2

引用 MPI 标准（3.7 节）:

在所有情况下，发送开始的调用 [意味着一个非阻塞发送] 都是局部的：它会立即返回而不管其他进程的状态。如果这个调用引起了一些系统资源被耗尽，那么它将会失败并且返回一个错误码。MPI 的高质量实现应该确保它仅发生在“病态的”情形。也就是说，一个 MPI 实现应该能够支持大量挂起的非阻塞操作 [P15]。

这意味着使用非阻塞调用是一种阻止死锁的可靠方法，因为 MPI 标准不允许在两个处理器上的一对匹配的发送和接收永远保持未完成状态。

习题 9.3

对于开边界条件，多达 12 个进程没有停滞期，这是因为每个子区域上面临通信的最大数随每个新的分解而改变。然而，我们将在 12 和 16 个进程间看到一个停滞期：从（3,2,2）变化到（4,2,2），没有新的子区域类型（考虑到通信特点）。如果在每个方向上至少有三个子

区域，那么将不再发生根本性改变，理想性能与实际性能之间的比率为固定值。这种情况至少含有 27 个进程（3,3,3）。

习题 9.4

最小的子区域大小（在 $3 \times 2 \times 2$=12 个处理器时）为 40×60^2，一次扫描大约为 1ms 的时间。这种情况下，50 μs 的延迟需要乘以 $k = 6$，因此聚合的延迟已经接近于 T_s 的 1/3。人们可以认为在高效实现的情形下，MPI_Reduce() 的开销是乒乓延迟的一个小的倍数（测量值在 50 μs 和 230 μs 之间，位于 1 ～ 12 个节点上，当然这也依赖于 N）。在拥有这些改进的情况下，模型能够很好地重现这些强扩展数据。

习题 9.5

进程 1 上的接收匹配进程 0 上的发送，但是后者可能抽不出空来调用 MPI_Send()。这是因为集合通信例程可能会将通信空间内的所有进程同步，但这不是必须的（当然，MPI_Barrier() 被定义用于同步）。如果广播是同步的，进程 1 上的接收会一直等待与它相匹配的发送而出现一个经典的死锁。 305

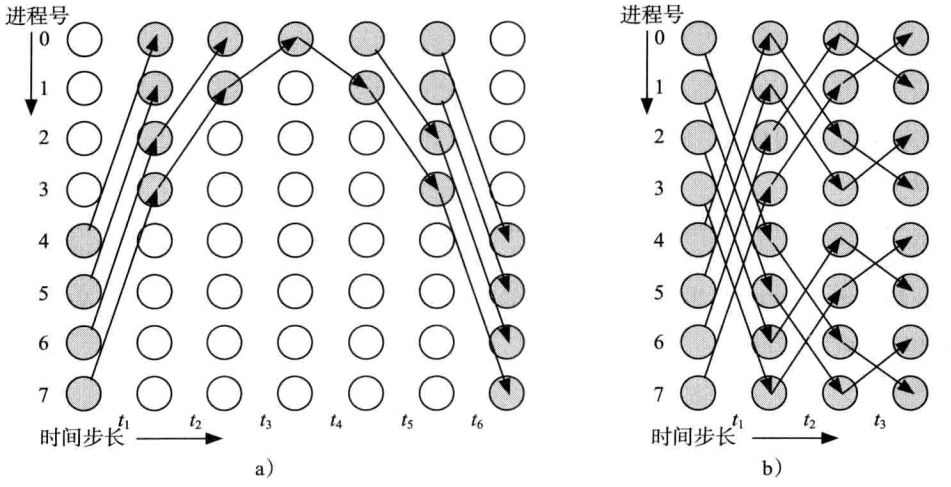


图 B-3 紧跟在一个广播 a）操作之后，“效仿”MPI_Allreduce 执行一个归约串行化了这两个操作，使得大量网络资源闲置。如果网络是非阻塞的，一个优化的“蝴蝶”型模式 b）会节省大量的时间

习题 10.1

传输消息的数量在两个例子中是一样的。因此，如果任意两个传输不能重叠，那么线性归约和对数归约的性能是一样的。总线网络（见 4.5.2 节）具有这样的性质。

甚至在一个完全非阻塞转换的网络上，如果使用静态路由，竞争也会出现（参见 4.5.3 节）。

习题 10.2

在广播之后进行归约需要它们的运行时间之和（见图 B-3a）。相反，一个最优化的 MPI_Allreduce() 操作可以通过利用网络中可用的并行点节省很多时间。

习题 10.3

在一个典型的“主 – 从”模式（见 5.2.2 节）中，如果消息大小小于急切的限制，那么工作进程的急切消息可能会淹没主进程。大多数 MPI 实现可以被配置为使用更大的缓冲区来存储急切消息。使用 MPI_Issend() 也可以作为一种选择。

习题 10.4

在数学上（见图 10-9），当 $N<11$ 时，“网柱”分解比立方体子区域更好。8 是带有至少
306 三个主要因素且小于 11 的唯一整数。如果我们将一个立方体分解为 $2\times2\times2$ 及 2×4 两种子区域，并且假设为周期边值条件，那么后者实际上有一个更小的区域切分表面。然而，这种差异对于开边界将会消失。

请注意，此处我们做了一点小小的欺骗：图 10-9 中的公式假设子区域总是立方体，并且网柱总是有一个正方形的基。这对于 $N=8$ 时并不成立。但是到目前为止，你可能已经意识到这仅仅是一个学术练习；三维分解带来更少的通信这条规则在实际应用中是成立的。

习题 10.5

$$B_{\text{eff}}(N, L, w, T_\ell, B)=\left[\frac{T_\ell N^{2/3}}{wL^2}+B^{-1}\right]^{-1} \tag{B-15}$$

延迟对带宽的影响随着 N 增加而增大。这个作业能被大的现场数据大小 w 和线性问题大小 L 部分地补偿。后面仅仅依赖于较大问题导致更好的可扩展性这个通常原则的另一个例子（这个例子中是由于减少的通信开销）。

注意，人们应该总是确保无论通信扮演什么样的角色，并行效率都是可以接受的。如果一个应用花费了它主要的时间用于通信，则“骑乒乓”曲线可能会是毫无意义的。人们甚至将不会考虑进一步增加 N，除非获得更多的内存。

习题 10.6

每个未解决的请求都需要一个独立的消息缓冲区，因此如果所有的晕环通信都要通过非阻塞 MPI 来处理，那么我们将需要 12 个中间缓冲区。由于这是一个表面积相对于体积的效应，因此额外的内存需求通常是微不足道的。此外，MPI_Wait() 必须被 MPI_Waitany() 或者 MPI_Waitsome() 所替换。

习题 10.7

```
1 call MPI_Isend(...)
2 call MPI_Irecv(...)
3 call MPI_Waitall(...)
```

一个可能有用的附加效果是，如果一组发送和接收操作同时处于未完成状态，那么 MPI 库可能会执行双全工传输。现今的 MPI 实现都是这么做的。

习题 10.8

如果所有的子区域具有相同的大小，那么内部的子区域将会执行得很慢。尽管它们的计算负载与所有其他区域并无差别，但它们必须花费更多时间用于通信，这导致了负载不均
307 衡。然而，如果进程数很大，则那些内部区域将占据主导（从数量上）。边界子区域是调速装置，并且只允许少量出现（见 5.3.9 节），因此这对于大规模问题没有影响。另一方面，若只有 $3\times3\times3=27$ 个进程，且通信开销成为问题，那么人们会考虑扩大边界子区域以获得更
308 好的负载均衡。

参考文献

基础文献

[S1] S. Goedecker and A. Hoisie. *Performance Optimization of Numerically Intensive Codes* (SIAM), 2001. ISBN 978-0898714845.

[S2] R. Gerber, A. J. C. Bik, K. Smith and X. Tian. *The Software Optimization Cookbook* (Intel Press), 2nd ed., 2005. ISBN 978-0976483212.

[S3] K. Dowd and C. Severance. *High Performance Computing* (O'Reilly & Associates, Inc., Sebastopol, CA, USA), 1998. ISBN 156592312X.

[S4] K. R. Wadleigh and I. L. Crawford. *Software Optimization for High-Performance Computing* (Prentice Hall), 2000. ISBN 978-0130170088.

[S5] W. Schönauer. *Scientific Supercomputing: Architecture and Use of Shared and Distributed Memory Parallel Computers* (Self-edition), 2000. http://www.rz.uni-karlsruhe.de/~rx03/book

[S6] T. G. Mattson, B. A. Sanders and B. L. Massingill. *Patterns for Parallel Programming* (Addison-Wesley), 2004. ISBN 978-0-321-22811-6.

[S7] D. H. Bailey. *Highly parallel perspective: Twelve ways to fool the masses when giving performance results on parallel computers.* Supercomputing Review **4(8)**, (1991) 54–55. ISSN 1048-6836. http://crd.lbl.gov/~dhbailey/dhbpapers/twelve-ways.pdf

并行编程

[P8] S. Akhter and J. Roberts. *Multi-Core Programming: Increasing Performance through Software Multithreading* (Intel Press), 2006. ISBN 0-9764832-4-6.

[P9] D. R. Butenhof. *Programming with POSIX Threads* (Addison-Wesley), 1997. ISBN 978-0201633924.

[P10] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism* (O'Reilly), 2007. ISBN 978-0596514808.

[P11] *The OpenMP API specification for parallel programming.* http://openmp.org/wp/openmp-specifications/

[P12] B. Chapman, G. Jost and R. van der Pas. *Using OpenMP* (MIT Press), 2007. ISBN 978-0262533027.

[P13] W. Gropp, E. Lusk and A. Skjellum. *Using MPI* (MIT Press), 2nd ed., 1999. ISBN 0-262-57132-3.

[P14] W. Gropp, E. Lusk and R. Thakur. *Using MPI-2* (MIT Press), 1999. ISBN 0-262-57133-1.

[P15] *MPI: A Message-Passing Interface Standard. Version 2.2*, September 2009. http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf

[P16] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek and V. Sunderam. *PVM: Parallel Virtual Machine* (MIT Press), 1994. ISBN 0-262-57108-0. http://www.netlib.org/pvm3/book/pvm-book.html

[P17] R. W. Numrich and J. Reid. *Co-Array Fortran for Parallel Programming.* SIGPLAN Fortran Forum **17(2)**, (1998) 1–31. ISSN 1061-7264.

[P18] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks and K. Warren. *Introduction to UPC and language specification.* Tech. rep., IDA Center for Computing Sciences, Bowie, MD, 1999. http://www.gwu.edu/~upc/publications/upctr.pdf

工具

[T19] *OProfile — A system profiler for Linux.* http://oprofile.sourceforge.net/news/

[T20] J. Treibig, G. Hager and G. Wellein. *LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments.* Submitted. http://arxiv.org/abs/1004.4431

[T21] *Intel VTune Performance Analyzer.* http://software.intel.com/en-us/intel-vtune

[T22] *PAPI: Performance Application Programming Interface.* http://icl.cs.utk.edu/papi/

[T23] *Intel Thread Profiler.* http://www.intel.com/cd/software/products/asmo-na/eng/286749.htm

[T24] D. Skinner. *Performance monitoring of parallel scientific applications*, 2005. http://www.osti.gov/bridge/servlets/purl/881368-dOvpFA/881368.pdf

[T25] O. Zaki, E. Lusk, W. Gropp and D. Swider. *Toward scalable performance visualization with Jumpshot.* International Journal of High Performance Computing Applications **13(3)**, (1999) 277–288.

[T26] *Intel Trace Analyzer and Collector.* http://software.intel.com/en-us/intel-trace-analyzer/

[T27] *VAMPIR - Performance optimization for MPI.* http://www.vampir.eu

[T28] M. Geimer, F. Wolf, B. J. Wylie, E. Ábrahám, D. Becker and B. Mohr. *The SCALASCA performance toolset architecture.* In: *Proc. of the International Workshop on Scalable Tools for High-End Computing (STHEC 2008)* (Kos, Greece), 51–65.

[T29] M. Gerndt, K. Fürlinger and E. Kereku. *Periscope: Advanced techniques for performance analysis.* In: G. R. Joubert *et al.* (eds.), *Parallel Computing: Current and Future Issues of High-End Computing (Proceedings of*

the International Conference ParCo 2005), vol. 33 of *NIC Series*. ISBN 3-00-017352-8.

[T30] T. Klug, M. Ott, J. Weidendorfer, and C. Trinitis. *autopin - Automated optimization of thread-to-core pinning on multicore systems*. Transactions on High-Performance Embedded Architectures and Compilers **3(4)**, (2008) 1–18.

[T31] M. Meier. *Pinning OpenMP threads by overloading pthread_create()*. http://www.mulder.franken.de/workstuff/pthread-overload.c

[T32] *Portable Linux processor affinity*. http://www.open-mpi.org/software/plpa/

[T33] *Portable hardware locality (hwloc)*. http://www.open-mpi.org/projects/hwloc/

计算机体系结构与设计

[R34] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach* (Morgan Kaufmann), 4th ed., 2006. ISBN 978-0123704900.

[R35] G. E. Moore. *Cramming more components onto integrated circuits*. Electronics **38(8)**, (1965) 114–117.

[R36] W. D. Hillis. *The Connection Machine* (MIT Press), 1989. ISBN 978-0262580977.

[R37] N. R. Mahapatra and B. Venkatrao. *The Processor-Memory Bottleneck: Problems and Solutions*. Crossroads **5**, (1999) 2. ISSN 1528-4972.

[R38] M. J. Flynn. *Some computer organizations and their effectiveness*. IEEE Trans. Comput. **C-21**, (1972) 948.

[R39] R. Kumar, D. M. Tullsen, N. P. Jouppi and P. Ranganathan. *Heterogeneous chip multiprocessors*. IEEE Computer **38(11)**, (2005) 32–38.

[R40] D. P. Siewiorek, C. G. Bell and A. Newell (eds.). *Computer Structures: Principles and Examples* (McGraw-Hill), 2nd ed., 1982. ISBN 978-0070573024. http://research.microsoft.com/en-us/um/people/gbell/Computer_Structures_Principles_and_Examples/

性能模型

[M41] J. Treibig, G. Hager and G. Wellein. *Multi-core architectures: Complexities of performance prediction and the impact of cache topology*. In: S. Wagner *et al.* (eds.), *High Performance Computing in Science and Engineering, Garching/Munich 2009* (Springer-Verlag, Berlin, Heidelberg). To appear. http://arxiv.org/abs/0910.4865

[M42] S. Williams, A. Waterman and D. Patterson. *Roofline: An insightful visual performance model for multicore architectures*. Commun. ACM **52(4)**, (2009) 65–76. ISSN 0001-0782.

[M43] P. F. Spinnato, G. van Albada and P. M. Sloot. *Performance modeling of distributed hybrid architectures*. IEEE Trans. Parallel Distrib. Systems **15(1)**, (2004) 81–92.

[M44] J. Treibig and G. Hager. *Introducing a performance model for bandwidth-limited loop kernels*. In: *Proceedings of PPAM 2009, the Eighth International Conference on Parallel Processing and Applied Mathematics, Wroclaw, Poland, September 13–16, 2009*. To appear.
http://arxiv.org/abs/0905.0792

[M45] G. M. Amdahl. *Validity of the single processor approach to achieving large scale computing capabilities*. In: *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, Spring Joint Computer Conference* (ACM, New York, NY, USA), 483–485.

[M46] J. L. Gustafson. *Reevaluating Amdahl's law*. Commun. ACM **31(5)**, (1988) 532–533. ISSN 0001-0782.

[M47] M. D. Hill and M. R. Marty. *Amdahl's Law in the multicore era*. IEEE Computer **41(7)**, (2008) 33–38.

[M48] X.-H. Sun and Y. Chen. *Reevaluating Amdahl's Law in the multicore era*. Journal of Parallel and Distributed Computing **70(2)**, (2010) 183–188. ISSN 0743-7315.

数值方法与软件库

[N49] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods* (SIAM), 1993. ISBN 978-0-898713-28-2.

[N50] C. L. Lawson, R. J. Hanson, D. R. Kincaid and F. T. Krogh. *Basic Linear Algebra Subprograms for Fortran usage*. ACM Transactions on Mathematical Software **5(3)**, (1979) 308–323. ISSN 0098-3500.

[N51] W. H. Press, B. P. Flannery, S. A. Teukolsky and W. T. Vetterling. *Numerical Recipes in FORTRAN 77: The Art of Scientific Computing (v. 1)* (Cambridge University Press), 2nd ed., September 1992. ISBN 052143064X.
http://www.nr.com/

优化技术

[O52] G. Wellein, G. Hager, T. Zeiser, M. Wittmann and H. Fehske. *Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization*. Annual International Computer Software and Applications Conference (COMPSAC09) **1**, (2009) 579–586. ISSN 0730-3157.

[O53] M. Wittmann, G. Hager and G. Wellein. *Multicore-aware parallel temporal blocking of stencil codes for shared and distributed memory*. In: *Workshop on Large-Scale Parallel Processing 2010 (IPDPS2010), Atlanta, GA, April*

23, 2010.
http://arxiv.org/abs/0912.4506

[O54] G. Hager, T. Zeiser, J. Treibig and G. Wellein. *Optimizing performance on modern HPC systems: Learning from simple kernel benchmarks.* In: *Proceedings of 2nd Russian-German Advanced Research Workshop on Computational Science and High Performance Computing, Stuttgart 2005* (Springer-Verlag, Berlin, Heidelberg).

[O55] A. Fog. *Agner Fog's software optmization resources.*
http://www.agner.org/optimize/

[O56] G. Schubert, G. Hager and H. Fehske. *Performance limitations for sparse matrix-vector multiplications on current multicore environments.* In: S. Wagner *et al.* (eds.), *High Performance Computing in Science and Engineering, Garching/Munich 2009* (Springer-Verlag, Berlin, Heidelberg). To appear.
http://arxiv.org/abs/0910.4836

[O57] D. J. Kerbyson, M. Lang and G. Johnson. *Infiniband routing table optimizations for scientific applications.* Parallel Processing Letters **18(4)**, (2008) 589–608.

[O58] M. Wittmann and G. Hager. *A proof of concept for optimizing task parallelism by locality queues.*
http://arxiv.org/abs/0902.1884

[O59] G. Hager, F. Deserno and G. Wellein. *Pseudo-vectorization and RISC optimization techniques for the Hitachi SR8000 architecture.* In: S. Wagner *et al.* (eds.), *High Performance Computing in Science and Engineering Munich 2002* (Springer-Verlag, Berlin, Heidelberg), 425–442.

[O60] D. Barkai and A. Brandt. *Vectorized multigrid poisson solver for the CDC Cyber 205.* Applied Mathematics and Computation **13**, (1983) 217–227.

[O61] M. Kowarschik. *Data Locality Optimizations for Iterative Numerical Algorithms and Cellular Automata on Hierarchical Memory Architectures* (SCS Publishing House), 2004. ISBN 3-936150-39-7.

[O62] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf and K. Yelick. *Optimization and performance modeling of stencil computations on modern microprocessors.* SIAM Review **51**, (2009) 129–159.

[O63] J. Treibig, G. Wellein and G. Hager. *Efficient multicore-aware parallelization strategies for iterative stencil computations.* Submitted.
http://arxiv.org/abs/1004.1741

[O64] M. Müller. *Some simple OpenMP optimization techniques.* In: *OpenMP Shared Memory Parallel Programming: International Workshop on OpenMP Applications and Tools, WOMPAT 2001, West Lafayette, IN, USA, July 30-31, 2001: Proceedings.* 31–39.

[O65] G. Hager, T. Zeiser and G. Wellein. *Data access optimizations for highly threaded multi-core CPUs with multiple memory controllers.* In: *Workshop*

on Large-Scale Parallel Processing 2008 (IPDPS2008), Miami, FL, April 18, 2008.
http://arxiv.org/abs/0712.2302

[O66] S. Williams, L. Oliker, R. W. Vuduc, J. Shalf, K. A. Yelick and J. Demmel. *Optimization of sparse matrix-vector multiplication on emerging multicore platforms.* Parallel Computing **35(3)**, (2009) 178–194.

[O67] C. Terboven, D. an Mey, D. Schmidl, H. Jin and T. Reichstein. *Data and thread affinity in OpenMP programs.* In: *MAW '08: Proceedings of the 2008 workshop on Memory access on future processors* (ACM, New York, NY, USA). ISBN 978-1-60558-091-3, 377–384.

[O68] B. Chapman, F. Bregier, A. Patil and A. Prabhakar. *Achieving performance under OpenMP on ccNUMA and software distributed shared memory systems.* Concurrency Comput.: Pract. Exper. **14**, (2002) 713–739.

[O69] R. Rabenseifner and G. Wellein. *Communication and optimization aspects of parallel programming models on hybrid architectures.* Int. J. High Perform. Comp. Appl. **17(1)**, (2003) 49–62.

[O70] R. Rabenseifner, G. Hager and G. Jost. *Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes.* In: D. E. Baz, F. Spies and T. Gross (eds.), *Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing, PDP 2009, Weimar, Germany, 18–20 Febuary 2009* (IEEE Computer Society). ISBN 978-0-7695-3544-9, 427–436.

[O71] G. Hager, G. Jost and R. Rabenseifner. *Communication characteristics and hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes.* In: *Proceedings of CUG09, May 4–7 2009, Atlanta, GA.*
http://www.cug.org/7-archives/previous_conferences/CUG2009/bestpaper/9B-Rabenseifner/rabenseifner-paper.pdf

[O72] H. Stengel. *Parallel programming on hybrid hardware: Models and applications.* Master thesis, Georg Simon Ohm University of Applied Sciences, Nuremberg, 2010.
http://www.hpc.rrze.uni-erlangen.de/Projekte/hybrid.shtml

[O73] M. Frigo and V. Strumpen. *Cache oblivious stencil computations.* In: *ICS '05: Proceedings of the 19th annual international conference on Supercomputing* (ACM, New York, NY, USA). ISBN 1-59593-167-8, 361–366.

[O74] M. Frigo and V. Strumpen. *The memory behavior of cache oblivious stencil computations.* J. Supercomput. **39(2)**, (2007) 93–112. ISSN 0920-8542.

[O75] T. Zeiser, G. Wellein, A. Nitsure, K. Iglberger, U. Rüde and G. Hager. *Introducing a parallel cache oblivious blocking approach for the lattice Boltzmann method.* Progress in CFD **8(1–4)**, (2008) 179–188.

大规模并行

[L76] A. Hoisie, O. Lubeck and H. Wasserman. *Performance and scalability analysis of teraflop-scale parallel architectures using multidimensional wavefront applications.* Int. J. High Perform. Comp. Appl. **14**, (2000) 330.

[L77] F. Petrini, D. J. Kerbyson and S. Pakin. *The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q.* In: *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing* (IEEE Computer Society, Washington, DC, USA). ISBN 1-58113-695-1, 55.

[L78] D. J. Kerbyson and A. Hoisie. *Analysis of wavefront algorithms on large-scale two-level heterogeneous processing systems.* In: *Proceedings of the Workshop on Unique Chips and Systems (UCAS2), IEEE Int. Symposium on Performance Analysis of Systems and Software (ISPASS), Austin, TX, 2006.*

应用

[A79] H. Fehske, R. Schneider and A. Weisse (eds.). *Computational Many-Particle Physics*, vol. 739 of *Lecture Notes in Physics* (Springer), 2008. ISBN 978-3-540-74685-0.

[A80] A. Fava, E. Fava and M. Bertozzi. *MPIPOV: A parallel implementation of POV-Ray based on MPI.* In: *Proc. Euro PVM/MPI'99*, vol. 1697 of *Lecture Notes in Computer Science* (Springer), 426–433.

[A81] B. Freisleben, D. Hartmann and T. Kielmann. *Parallel raytracing: A case study on partitioning and scheduling on workstation clusters.* In: *Proc. 30th International Conference on System Sciences 1997, Hawaii* (IEEE), 596–605.

[A82] G. Wellein, G. Hager, A. Basermann and H. Fehske. *Fast sparse matrix-vector multiplication for TFlops computers.* In: J. Palma *et al.* (eds.), *High Performance Computing for Computational Science — VECPAR2002, LNCS 2565* (Springer-Verlag, Berlin, Heidelberg). ISBN 3-540-00852-7, 287–301.

[A83] G. Hager, E. Jeckelmann, H. Fehske and G. Wellein. *Parallelization strategies for density matrix renormalization group algorithms on shared-memory systems.* J. Comput. Phys. **194(2)**, (2004) 795–808.

[A84] M. Kinateder, G. Wellein, A. Basermann and H. Fehske. *Jacobi-Davidson algorithm with fast matrix vector multiplication on massively parallel and vector supercomputers.* In: E. Krause and W. Jäger (eds.), *High Performance Computing in Science and Engineering '00* (Springer-Verlag, Berlin, Heidelberg), 188–204.

[A85] H. Fehske, A. Alvermann and G. Wellein. *Quantum transport within a background medium: Fluctuations versus correlations.* In: S. Wagner *et al.* (eds.), *High Performance Computing in Science and Engineering, Garching/Munich 2007* (Springer-Verlag, Berlin, Heidelberg). ISBN 978-3-540-69181-5, 649–668.

[A86] T. Pohl, F. Deserno, N. Thürey, U. Rüde, P. Lammers, G. Wellein and T. Zeiser. *Performance evaluation of parallel large-scale lattice Boltzmann applications on three supercomputing architectures.* In: *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing.* http://www.sc-conference.org/sc2004/schedule/index.php?module=

Default\&action=ShowDetail\&eventid=13\#2

[A87] C. Körner, T. Pohl, U. Rüde, N. Thürey and T. Zeiser. *Parallel Lattice Boltzmann Methods for CFD Applications*. In: *Numerical Solution of Partial Differential Equations on Parallel Computers* (Springer-Verlag, Berlin, Heidelberg). ISBN 3-540-29076-1, 439–465.

[A88] G. Allen, T. Dramlitsch, I. Foster, N. T. Karonis, M. Ripeanu, E. Seidel and B. Toonen. *Supporting efficient execution in heterogeneous distributed computing environments with Cactus and Globus*. In: *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing* (ACM, New York, NY, USA). ISBN 1-58113-293-X, 52–52.

[A89] G. Hager, H. Stengel, T. Zeiser and G. Wellein. *RZBENCH: Performance evaluation of current HPC architectures using low-level and application benchmarks*. In: S. Wagner *et al.* (eds.), *High Performance Computing in Science and Engineering, Garching/Munich 2007* (Springer-Verlag, Berlin, Heidelberg). ISBN 978-3-540-69181-5, 485–501.
http://arxiv.org/abs/0712.3389

[A90] D. Kaushik, S. Balay, D. Keyes and B. Smith. *Understanding the performance of hybrid MPI/OpenMP programming model for implicit CFD codes*. In: *Parallel CFD 2009 - 21st International Conference on Parallel Computational Fluid Dynamics, Moffett Field, CA, USA, May 18–22, 2009, Proceedings*. ISBN 978-0-578-02333-5, 174–177.

C++ 文献

[C91] A. Fog. *Optimizing software in C++: An optimization guide for Windows, Linux and Mac platforms*.
http://www.agner.org/optimize/optimizing_cpp.pdf

[C92] D. Bulka and D. Mayhew. *Efficient C++: Performance Programming Techniques* (Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA), 1999. ISBN 0-201-37950-3.

[C93] S. Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition)* (Addison-Wesley Professional), 2005. ISBN 0321334876.

[C94] S. Meyers. *More Effective C++: 35 New Ways to Improve Your Programs and Designs* (Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA), 1995. ISBN 020163371X.

[C95] S. Meyers. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library* (Addison-Wesley Longman Ltd., Essex, UK, UK), 2001. ISBN 0-201-74962-9.

[C96] T. Veldhuizen. *Expression templates*. C++ Report **7(5)**, (1995) 26–31.
http://ubiety.uwaterloo.ca/~tveldhui/papers/Expression-Templates/exprtmpl.html

[C97] J. Härdtlein, A. Linke and C. Pflaum. *Fast expression templates*. In: V. S. Sunderam, G. D. van Albada, P. M. A. Sloot and J. Dongarra (eds.), *Computational Science - ICCS 2005, 5th International Conference, Atlanta, GA,*

USA, May 22–25, 2005, Proceedings, Part II. 1055–1063.

[C98] A. Aue. *Improving performance with custom pool allocators for STL.* C/C++ Users's Journal , (2005) 1–13.
http://www.ddj.com/cpp/184406243

[C99] M. H. Austern. *Segmented iterators and hierarchical algorithms.* In: M. Jazayeri, R. Loos and D. R. Musser (eds.), *International Seminar on Generic Programming, Dagstuhl Castle, Germany, April 27 - May 1, 1998, Selected Papers*, vol. 1766 of *Lecture Notes in Computer Science* (Springer). ISBN 3-540-41090-2, 80–90.

[C100] H. Stengel. *C++-Programmiertechniken für High Performance Computing auf Systemen mit nichteinheitlichem Speicherzugriff unter Verwendung von OpenMP.* Diploma thesis, Georg-Simon-Ohm University of Applied Sciences Nuremberg, 2007.
http://www.hpc.rrze.uni-erlangen.de/Projekte/numa.shtml

[C101] C. Terboven and D. an Mey. *OpenMP and C++.* In: *Proceedings of IWOMP2006 — International Workshop on OpenMP, Reims, France, June 12–15, 2006.*
http://iwomp.univ-reims.fr/cd/papers/TM06.pdf

[C102] British Standards Institute. *The C++ Standard: Incorporating Technical Corrigendum 1: BS ISO* (John Wiley & Sons, New York, London, Sydney), 2nd ed., 2003. ISBN 0-470-84674-7.

[C103] M. H. Austern. *What are allocators good for?* C/C++ Users's Journal, December 2000.
http://www.ddj.com/cpp/184403759

专业文档

[V104] *Intel 64 and IA-32 Architectures Optimization Reference Manual* (Intel Press), 2009.
http://developer.intel.com/design/processor/manuals/248966.pdf

[V105] *Software Optimization Guide for AMD64 Processors* (AMD), 2005.
http://support.amd.com/us/Processor_TechDocs/25112.PDF

[V106] *Software Optimization Guide for AMD Family 10h Processors* (AMD), 2009.
http://support.amd.com/us/Processor_TechDocs/40546-PUB-Optguide_3-11_5-21-09.pdf

[V107] A. J. C. Bik. *The Software Vectorization Handbook: Applying Intel Multimedia Extensions for Maximum Performance* (Intel Press), 2004. ISBN 978-0974364926.

[V108] *Hyper-Threading technology.* Intel Technology Journal **6(1)**, (2002) 1–66. ISSN 1535766X.

[V109] R. Gerber and A. Binstock. *Programming with Hyper-Threading Technology* (Intel Press), 2004. ISBN 0-9717861-4-3.

[V110] *SUPER-UX Performance Tuning Guide* (NEC Corporation), 2006.

[V111] *Optimizing Applications on Cray X1 Series Systems* (Cray Inc.), 2007.

[V112] *Intel C++ intrinsics reference*, 2007.
http://www.intel.com/cd/software/products/asmo-na/eng/347603.htm

[V113] W. A. Triebel, J. Bissell and R. Booth. *Programming Itanium-Based Systems* (Intel Press), 2001. ISBN 978-0970284624.

[V114] N. Adiga *et al. An overview of the BlueGene/L supercomputer.* In: *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing* (IEEE Computer Society Press, Los Alamitos, CA, USA), 1–22.

[V115] IBM Journal of Research and Development staff. *Overview of the IBM Blue Gene/P project.* IBM J. Res. Dev. **52(1/2)**, (2008) 199–220. ISSN 0018-8646.

[V116] C. Sosa and B. Knudson. *IBM System Blue Gene Solution: Blue Gene/P Application Development.* IBM Redbooks. 2009. ISBN 978-0738433332.
http://www.redbooks.ibm.com/Redbooks.nsf/RedbookAbstracts/sg247287.html

[V117] *Cray XT5 Supercomputer.*
http://www.cray.com/Products/XT5.aspx

网站和在线资源

[W118] *Standard Performance Evaulation Corporation.*
http://www.spec.org/

[W119] J. D. McCalpin. *STREAM: Sustainable memory bandwidth in high performance computers.* Tech. rep., University of Virginia, Charlottesville, VA, 1991-2007. A continually updated technical report.
http://www.cs.virginia.edu/stream/

[W120] J. Treibig. *Likwid: Linux tools to support programmers in developing high performance multi-threaded programs.*
http://code.google.com/p/likwid/

[W121] *Top500 supercomputer sites.*
http://www.top500.org

[W122] *HPC Challenge Benchmark.*
http://icl.cs.utk.edu/hpcc/

[W123] A. J. van der Steen and J. J. Dongarra. *Overview of recent supercomputers*, 2008.
http://www.euroben.nl/reports/web08/overview.html

[W124] *Intel MPI benchmarks.*
http://software.intel.com/en-us/articles/intel-mpi-benchmarks/

[W125] *MPICH2 home page.*
http://www.mcs.anl.gov/research/projects/mpich2/

[W126] *OpenMPI: A high performance message passing library.*

http://www.open-mpi.org/

[W127] *Intel MPI library.*
http://software.intel.com/en-us/intel-mpi-library/

[W128] *MPI forum.*
http://www.mpi-forum.org

计算机历史

[H129] R. Rojas and U. Hashagen (eds.). *The First Computers: History and Architectures* (MIT Press, Cambridge, MA, USA), 2002. ISBN 0262681374.

[H130] K. Zuse. *The Computer — My Life* (Springer), 1993. ISBN 978-3540564539.

[H131] P. E. Ceruzzi. *A History of Modern Computing* (MIT Press), 2nd ed., 2003. ISBN 978-0262532037.

其他

[132] S. E. Raasch and S. K. Reinhardt. *The impact of resource partitioning on SMT processors.* In: *International Conference on Parallel Architectures and Compilation Techniques, PACT 2003* (IEEE Computer Society, Los Alamitos, CA, USA). ISSN 1089-795X, 15.

[133] N. Anastopoulos and N. Koziris. *Facilitating efficient synchronization of asymmetric threads on hyper-threaded processors.* In: *IEEE International Symposium on Parallel and Distributed Processing (IPDPS) 2008.* ISSN 1530-2075, 1–8.

[134] J. D. McCalpin. *Memory bandwidth and machine balance in current high performance computers.* IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, December 1995.
http://tab.computer.org/tcca/NEWS/DEC95/dec95_mccalpin.ps

[135] D. Monniaux. *The pitfalls of verifying floating-point computations.* ACM Trans. Program. Lang. Syst. **30(3)**, (2008) 1–41. ISSN 0164-0925.
http://arxiv.org/abs/cs/0701192

[136] M. Bull. *Measuring synchronization and scheduling overheads in OpenMP.* In: *First European Workshop on OpenMP — EWOMP 99, Lund University, Lund, Sweden, Sep 30–Oct 1, 1999.*
http://www.it.lth.se/ewomp99/papers/bull.pdf

[137] R. Thakur, R. Rabenseifner and W. Gropp. *Optimization of collective communication operations in MPICH.* Int. J. High Perform. Comp. Appl. **19(1)**, (2005) 49–66.

[138] B. Goglin. *High Throughput Intra-Node MPI Communication with Open-MX.* In: *Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP2009)* (IEEE Computer Society Press, Weimar, Germany), 173–180.
http://hal.inria.fr/inria-00331209

[139] A. Kleen. *Developer Central Open Source: numactl and libnuma.*
http://oss.sgi.com/projects/libnuma/

索　引

索引中的页码为英文原书页码，与书中页边标注的页码一致。

A

accelerators（加速器），104，141
affinity（亲和性），26，174，277
  mask（～掩码），280
aggregation（聚合），248
aliasing（别名），54
alignment（队列），50
allocator class template（分配器类模板），199
Amdahl，Gene（Amdahl，因子），124
Amdahl's Law（Amdahl 定律），124，130，166，275，290，298
anisotropy（异性），26，100，103，158，260
architectural state（架构状态），26
arrays（数组）
  multidimensional（多维～），69
assembly code（汇编代码），56
asynchronous communication（异步通信），250

B

balance（平衡），63
  analysis（～分析），31，63
  code（～码），66
  machine（～机），63
bandwidth（带宽），4，15
  effective（有效～），105
bank busy time（bank 忙碌时间），30
basic block（基本块），38
batch system（批处理系统），207
benchmarking（基准测试）
  and time measurement（～与时间测量），5
  applications（应用～），7
  low-level（低层次～），4
big fat lock，178
BLAS（BLAS 库），85
branch（分支）
  elimination（～消除），48
  misprediction（～误测），26，43，48，176
  prediction（～预测），48
buffer cache（缓冲区 cache），194-197
bulk-synchronous communication（整体同步通信），275
bus network（总线网络），109，306
butterfly profile（蝶形轮廓），*see* profile，callgraph

C

C++
  and ccNUMA（～和 ccNUMA），197-201
  optimizations（～优化），56-61
C-style arrays（C 风格的数组），62，199
cache（高速缓存），3，4，8，15
  associativity（～结合性），77
  coherence（～一致性），97，189
    protocol（～协议），*see* MESI
  direct-mapped（～直接映射），19
  directory（～目录），98
  effective size（有效～尺寸），20，78
  fully associative（全相联～），18
  group（～　组），25，99，104，131，174，177，184，246，260，271，279
  hit（～命中），15
    ratio（～命中率），17
  levels（多级～），15
  line（～行），16，180
    replacement strategy（～行替换策略），18
    zero（～行 0），67，254
  miss（～未命中），15
  reuse（～重用）
    ratio（～重用率），16
  set-associative（～组相联），19

thrashing（～抖动），19，77
unified（～一致性），15
way（～快取通道），20
write-back（cache 写回），17
write-through（cache 写通），75
cache line zero（cache 行 0），18
cache-bound（cache 关键型），17
CAF（CAF 语言），xvii，204
call by value（按值调用），146
capacity miss（容量失效），28
ccNUMA，xviii，97，100，185-201
allocator（～分配符），199
and C++（～和 C++），197
contention problem（～的竞争问题），101
locality domain（～本地域），100，183，246，255
locality problem（～位置问题），101，174
memory mapping（～内存映射），186
page placement（～页替换），*see* ccNUMA memory mapping
CFD，xv，120
chaining（链），30
checkpoint/restart（检查点 / 重启），205
CISC，8
clusters（簇），102
column major order（以列为主的顺序），70，74，249
commodity clusters（商品化集群），1，103，104
compiler（编译器），2
directives（～指令），50，51，81
intrinsics（～内部），51
logs（～日志），55，81
complexity management（复杂度管理），56
computational（计算）
domain（～区域），117
grid（～网格），117
intensity（～密集度），66
conflict miss（冲突失效），19
Connection Machine（连接机），14，96
contention（竞争），26，101，110，158，243，250
copy constructor（复制构造函数），57，60，147
core（核），24
core ID（核 ID），277
cost model（成本模型），130，141
CPU，1，24
CPU time（CPU 时间），5
Cray 1，28
Cray XT，112，250，251，255
critical path（关键路径），40
crossbar switch（纵横开关），100，110
CRS，87

D

deadlock（死锁），149，150，211-213，216，229，233，240，261，305
denormals（非正规的），54
diminishing returns（收益递减），127
distributed memory（分布式内存），102
domain decomposition（区域分解），117，129，244
DRAM gap（DRAM 差距），15，65，116

E

Eager protocol（Eager 协议），239，241，260，275
Eckert，Presper，1
EDVAC，1
EPIC，91
event timeline（时间事件线）
MPI（MPI 的～），236
OpenMP（OpenMP 的～），165
eviction（替换），15
exclusive time（专有时间），38
expression templates（表达式模板），59

F

false sharing（伪共享），99，179
first touch policy（首次访问政策），186
floating-point（浮点）
accuracy（～准确性），54，149
units（～单元），2
flop（浮点运算），4
FMA，*see* multiply-add（FMA
fooling the masses（愚弄大众），141
frontside bus（前端总线），*see* FSB
FS cache（前端高速缓存），*see* buffer cache

FSB，99，156，183
functional decomposition（功能性解构），120

G

gather/scatter（聚集 / 发散），33
Gauss-Seidel algorithm（Gauss-Seidel 算法），273
parallel（并行～），158
ghost layers（影子层），117，129，224，242，244，246
  multilevel（多层～），248
gprof，38
GPU，96
granularity（粒度），134，138
Gustafson's Law（Gustafson 定律），125

H

halo layers（光晕层），*see* ghost layers
hardware thread（硬件线程），277
hierarchical systems（多级递阶系统），*see* hybrid systems
hot spots（热点），37
HPC，xvi
HPF，204
hwloc，284
hybrid（混合）
  MPI/OpenMP programming（～ MPI/OpenMP 编程），264
  programming（～程序设计），xviii，247，252
  systems（～系统），103，252
Hyper-Threading（超线程技术），*see* threading
hyperplane（超平面），301
HyperTransport（超传输），25，100，113，114，186，257

I

IBM Blue Gene（IBM 蓝色基因），112，137
ILLIAC IV，298
ILP，*see* instruction-level parallelism
IMB suite（IMB 套件），106，255
in-order execution（顺序执行），43
inclusive time（共享时间），39
InfiniBand，105，107，108
injection bandwidth（注入带宽），113
inlining（代码嵌入），39，52，58
instruction（指令）
  cache（～ cache），15
  queues（～队列），3
  stream（～流），*see* thread
  throughput（～吞吐量），7，27
instruction-level parallelism（指令级并行），7，8，14
instrumentation（工具）
  automatic（自动化～），38
  manual（～手册），45
integer units（整数单元），2
intranode communication（内点通信），253-260
IPM，235
iterators（迭代器），61
  segmented（段～），61

J

Jacobi algorithm（Jacobi 算法），71，117
  hybrid（混合型～），264，267
  parallel（并行～），156，224
JDS，88

L

L2 group（二级缓存），*see* cache group
laggers（滞后），137，307
latency（延迟），15，20
  of network（网络～），105，110，129
lazy construction（惰性构建），59
LD，*see* ccNUMA locality domain
libnuma（（函数）libnuma），285
lightspeed（光速），66
LIKWID tools（LIKWID 工具）
  affinity（～亲和性），281
  performance counters（～性能计数器），44
  topology（～拓扑），279
LINPACK，95
Linux，xvi，279
list vector（列表向量），34
load（负载）
  balancing（～均衡），118，304
  imbalance（～不均衡），119-121，137，151，152，

166，172，182
load/store units（存储单元），3
locality（局部性）
of reference（引用～），16
spatial（空间～），17，75
temporal（时间～），16
logical processor（逻辑处理器），27
loop（循环）
blocking（～阻塞），82，291
fusion（～合并），79
interchange（～交换），77
nest（～嵌套），79
parallelism（～并行），116，147
peeling（～脱皮），89，293
remainder（～剩余），49，80
skewing（～扭曲），301
splitting（～分裂），55
stripmining（～拆藏），30
unroll and jam（～展开和隐藏），81 93，292
unrolling（～展开），49，80，93，292
loop-carried dependencies（循环体依赖性），12，158，161，163，273
and SIMD（～与 SIMD），62
LRU，18，85
LUp. 73

M

`malloc()`，59，199
mapping problem（映射问题），246
mask（屏蔽）
registers（～存储器），32
master-worker（主从方式），120
matrix（矩阵），69
transpose（～转置），74
Mauchly，John，1
medium-grained parallelism（中等粒度并行），116
memory（内存）
affinity（～关联），277
bandwidth（～带宽），15，97
bus（～总线），100
latency（～延迟），15，97，176
memory page（内存页），76，295
large（大～），295
round-robin placement（～循环置换算法），192
table（～表），186，189
memory pages（内存页）
placement（置换），185
memory-bound（内存关键），17
MESI，98
message envelope（消息信封），239
message passing（消息传递），203
interface（～接口），*see* MPI
μ-ops（微操作），8
MIMD，96
MISD，96
MLUP，*see* LUP
Monte Carlo method（Monte Carlo 方法），162
Moore's Law（Moore 法则），7，15，23，96
MPI，102，106，117，203
benchmarks（～基准），105
blocking communication（～阻塞通信），209，216
collective communication（～集群通信），213
communicator（～通信域），206
data types（～数据类型），207
derived types（～派生类型），207，248
message tag（MPI 消息标签），207
nonblocking communication（～非阻塞通信），216，250
point-to-point communication（～点对点通信），207
profiling（～剖析），235-239
interface（～接口），235
progress（～进程），233，250，274
rank（～序列），204
reduction（～归约），211
request handle（～请求句柄），217
synchronous communication（～同步通信），209
virtual topologies（～虚拟拓扑），220
wildcards（～通配符），209
wrapper scripts（～封装脚本），205
`MPI_Allreduce ()`，260，306
`MPI_Alltoall ()`，244
`MPI_Barrier()`，213
`MPI_Bcast ()`，213
`MPI_Bsend ()`，213
`MPI_Cart_coords ()`，222

MPI_Cart_create (), 221, 246
MP I_Cart_rank (), 222
MPI_Cart_shi ft (), 223
MPI_Comm_rank (), 206
MPI_Comm_size (), 206
MPI_Comm_world, 206
MPI_Datatype, 248
MPI_Dims_create (), 222, 244
MPI_Finalize (), 206
MPI_Get_count (), 209
MPI_I nit (), 206
MPI_Init_thread (), 269
MPI_irecv (), 217, 250
MPI_Isend (), 217, 250
MPI_Issend (), 239
MPI_PROC_NULL, 223
MPI_Recv (), 208
MPI_Reduce (), 214, 253
MPI_Send (), 208, 249
MPI_Sendrecv (), 213, 242, 261
MPI_Sendrecv_replace (), 213, 242
MPI_Ssend (), 212, 241
MPI_Test (), 217
MPI_THREAD_XXXX, 268
MPI_Type_commit (), 249
MPI_Type_free (), 249
MPI_Type_vector (), 249
MPI_Type_XXXXX, 248
MPI_Wait (), 217
mpirun, 206, 281
MPMD, 119, 203
multicore (多核), 156
  awareness (～感知), 26
  processor (～处理器), 3, 9, 23 109, 255
  topology (～拓扑), 279
multilayer halo (多层重叠区), 248
multiply-add (累加 (指令)), 56, 74

N

NEC
  SX-8 (NEC SX), 6, 31
  SX-9 (NEC SX), 29, 100
network (网络), 102
  bisection bandwidth (～对分带宽), 108
  contention (～线路竞争), 243, 246
  diameter (～直径), 110
  fat-tree (～胖树形), 110
  hybrid (～混合), 113
  latency (～延迟), 110
  mesh (～多跳), 112
  nonblocking (～无阻塞), 103
  switched (～交换), 110
  topology (～拓补), 104, 246
node (节点), 102
nontemporal stores (无暂存存储), 18, 67, 69, 254, 292
NORMA, 103
NRU, 18
numactl, 284
NUMALink, 101
numatools, 190

O

OMP_DYNAMIC, 156
omp_get_num_threads (), 145
omp_get_thread_num (), 145
omp_in_parallel (), 199
OMP_NUM_THREADS, 145, 281
OMP_STACKSIZE, 156
OpenMP, 97, 117, 143
  barrier (～栅栏), 150, 159, 170
  implicit (内含的), 150, 154, 170, 176
  chunksize (～数据块大小), 151, 174, 189, 193
  COLLAPSE clause (～ COLLAPSE 子句), 172
  conditional compilation (～条件编译), 154
  COPYIN clause (～ COPYIN 子句), 147
  critical region (～临界区域), 149, 177
  data scoping (～数据作用域), 146
  DO, 147
  F90 module (～ F90 模块), 145
  FIRSTPRIVATE, 147, 154
  FLUSH, 155
  for, 148
  include files (～包含文件), 145
  LASTPRIVATE, 163

lock（～锁），150，178
loop scheduling（～循环调度），151
master thread（～主线程），144
NOWAIT clause（～ NOWAIT 子句），150，170
NUM_THREADS clause（～ NUM_THREADS 子句），169
overhead（～开销），158，168，175
parallel region（～并行域），144
continuous（连续），171
PRIVATE clause（～ PRIVATE 子句），147
profiling（～性能分析），165
reduction（～归约），150，178，180
schedule（～任务调度），188
SCHEDULE clause（～ SCHEDULE 子句），151
sentinel（～哨兵），145
SINGLE，153，154
TASK construct（～ TASK 结构），154
task scheduling point（～任务调度点），154
tasking（～任务分配），153
thread（～线程），144
thread ID（～线程 ID），145
THREADPRIVATE clause（～ THREADPRIVATE 子句），147
worksharing directives（～工作分摊指令），147
operator new（new 运算符），198，202
optimization（优化）
by compiler（通过编译器～），13，41，51-56
for C++（为 C++ ～），56
orphaned directives（孤立指令），148
OS jitter（操作系统抖动），140
out-of-core techniques（核外技术），115
out-of-order execution（乱序执行），8，14
oversubscription（过载），111

P

padding（填充），78，180
parallel（并行）
computing（～计算），95
efficiency（～效率），125
file system（～文件系统），116
parallelism（并行度）
data（数据～），116
functional（功能～），119
parallelization（并行化）
automatic（自动～），143
incremental（增量～），165
PCI bus（PCI 总线），109
peak performance（性能峰值），3
performance counters（性能计数器），41，55
derived metrics（派生度量～），43
multiplexing（复用～），44
performance models（性能模型），xviii
bandwidth-based（基于带宽～），63-67，71-74
parallel（并行～），123-137，175，231-232，234
PingPong，105，231，239，255
ride（～ ride），248，260
pinning（阻塞），155，186，277
pipeline（管道）
bubbles（气泡～），11，26，43
depth（深度～），10，11
flush（清空～），48
latency（延迟～），10
multitrack（多道～），29
stalls（停顿～），8，12，43
throughput（吞吐量～），10
wind-down（逐渐减少～），10
wind-up（逐渐增加～），10
pipeline parallel processing（流水线并行处理），*see* wave-front parallelization
pipelining（流水线），7，9
placement new（重载 new），197，201
PLPA，283
POSIX threads（POSIX 线程），143，145，281
power（功耗）
dissipation（损耗～），23
envelope（包络～），23
power-performance dilemma（功耗 - 性能困境），9，23，137
prefetch（预取），20，65，174
in hardware（在硬件～），21，303，292
outstanding（在外～），21
in software（在软件～），21
printf debugging（printf 调试），52
privatization（私有化），146，178，180
processor（处理器），24
profile（配置文件）
callgraph（函数调用关系图～），39

flat（平面～），38
profiling（设置文件）
function-based（基于函数～），38
and inlining（～和内联），39
line-based（基于行～），40
OpenMP（OpenMP ～），165
scalar（标量～），37
programming model（编程模型），102
pthread_create ()，281

Q

QuickPath，25，100，114

R

race condition（竞争条件），148，149，159，170
rand ()，155
ray tracing（光线跟踪），120
reduction（归约），150，178，180，211，213
register（寄存器），3，15
pressure（～不足），52，55，81，82
spill（～溢出），55，81
rendezvous protocol（集合点协议），239，241，254
reorder buffer（重排序缓冲区），8
return value optimization（返回值优化），58
ring shift（环形移位），211，240
RISC，8
root rank（根阶层），214
routing（路由）
adaptive（自适应～），112
static（静态～），111
row major order（行优先次序），70

S

sampling（采样），38
scalability（可扩展性），121，122
scaling（扩展）
baseline（基线～），130，131，183，192
strong（强～），123
superlinear（超线性～），130，184
weak（弱～），123，124
sched_setaffinity ()，283
Schönauer，Willi，5
scope（作用域），52
sequential equivalence（等效性时序），149
serialization（串行化），121
SGI Altix，43，101，183
shared memory（共享内存），97
shared resources（共享资源），121
shepherd（看管）
process（～进程），281
threads（～线程），281
SIMD，8，14，28，92，96
alignment（～阵列），50
vectorization（～向量化），49，54，61，62，287，294
single-copy transfer（单副本转移），254
single-copy transfer（单副本转移），256
SISD. 2，96
slow computing（缓慢计算），132
SMP，97
SMT，*see* threading
snoop（监听），98
tilter（～滤波器），99
socket（套接字），24
software pipelining（软件流水化），12，48，65
sparse（稀疏）
matrix（～矩阵），86
MVM（～ MVM），86
spatial blocking（空间封锁），292
speeders，137，307
speedup（加速器），120，123
spin-waiting loop（spin-waiting 循环），28
SPMD，116，203
SSE，8，50
stack（栈），288
stall cycles（停顿周期），*see* pipeline stalls
static construction（静态结构），59
stencil（模板），71
STL，199
storage order（存储顺序），*see* arrays，multidimensional
stored-program computer（存储程序计算机 ），1
STREAM，67，73
streaming（流），16
strength reduction（强度衰减），46

subnormals（次正规），*see* denormals
superlinear speedup（超线性加速比），130，184
superscalar（超标量）
  architecture（～体系结构），8
  processors（～处理器），13
sweeper code（sweeper 代码），196
switch（交换机），110
  hierarchy（层次～），110
  leaf（叶节点～），111
  nonblocking（非阻塞～），110
  spine（spine ～），111
synchronization（同步），28，211，213
  point（～点），137，150

T

tabulation（表格），47，294
task mode（任务方式），265
`taskset`，280
TBB，143
temporal blocking（时序阻塞），292
temporaries（临时变量），56
Thinking Machines（智能机），96
thread（线程），27
  affinity（～亲缘性），277
  pinning（～绑定），155
  placement（～置换），155，174
  safety（～安全），149，155，199，268，298
threading（线程），26，277
throughput mode（全模式），297
TLB，75，93
  misses（失配），76，287，292，303
Top500，95
Turing，Alan，1

U

UMA，97
UPC，xvii，204

V

vector（向量）
  computers（计算机～），1，28
  gather（采集～），33
  length（长度～），29
  registers（寄存器～），29
  scatter（离散度～），33
  triad（三元组～），5，17，67，175，187，192，202
vector mode（向量方式），264，275
`vector<>` template（vector<> 模板），60，199
vectorization（向量化），30
`volatile`（volatile 修饰符），155
von Neumann Bottleneck（冯·诺依曼瓶颈），2

W

wallclock time（墙上时间），5
wavefront parallelization（波阵面并行化），159
WC buffer（WC 缓冲区），*see* write combine buffer
wind-down（逐渐减少），10，159
wind-up（逐渐增加），10，159
wirespeed（线速），103
working set（工作集），47
write（写）
  hit（～命中），17
  miss（～缺失），18
write allocate（写分配），18，67，69，71，254，289
  in-cache（在 cache 中～），75
write combine buffer（写归并缓冲区），18

X

x86，xvi，279

Z

Zuse，Konrad，xv